Correctness, Efficiency and Durability of Concurrent and Distributed Systems

Gal Sela

Correctness, Efficiency and Durability of Concurrent and Distributed Systems

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Gal Sela

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tammuz 5784 Haifa July 2024

This research was carried out under the supervision of Prof. Erez Petrank, in the Faculty of Computer Science.

The author of this thesis states that the research, including the collection, processing and presentation of data, addressing and comparing to previous research, etc., was done entirely in an honest way, as expected from scientific research that is conducted according to the ethical standards of the academic world. Also, reporting the research and its results in this thesis was done in an honest and complete manner, according to the same standards.

The results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's doctoral research period:

Gal Sela and Erez Petrank. Concurrent size. *PACMPL*, 6(OOPSLA2), 2022.

Gal Sela, Maurice Herlihy, and Erez Petrank. Brief announcement: linearizability: a typo. In *PODC*, 2021.

Naama Ben-David, Gal Sela, and Adriana Szekeres. The FIDS theorems: tensions between multinode and multicore performance in transactional systems. In *DISC*, 2023.

Gal Sela and Erez Petrank. Durable queues: the second amendment. In *SPAA*, 2021.

Keren Censor-Hillel, Shir Cohen, Ran Gelles, and Gal Sela. Distributed computations in fully-defective networks. In *PODC*, 2022.

Keren Censor-Hillel, Shir Cohen, Ran Gelles, and Gal Sela. Distributed computations in fully-defective networks. *DC*, 2023.

Acknowledgements

I cannot thank my advisor, Prof. Erez Petrank, enough. Erez is one of the nicest people I have had the pleasure of meeting, as anyone who has met him will attest, and every meeting with Erez is not only educational but also enjoyable. I enjoyed learning about concurrency from him and working together on fascinating problems. Erez guided me both academically and professionally, and beyond the knowledge I gained and the development I underwent as a researcher through our joint work, he also introduced me to many people in the field. But Erez was much more than an advisor; he also cared for me personally, considered my needs, and helped me beyond anything I could have asked for. This is even before mentioning the joint trip with Erez and Yael after the conference in New Zealand, and the conference where my suitcase did not arrive, and without Erez, I would not have even had pajamas…

I thank my research partners, Keren Censor-Hillel, Ran Gelles, Shir Cohen, Naama Ben-David, Adriana Szekeres, Maurice Herlihy, Samuel Thomas, Tali Moreshet, Iris Bahar, and Hen Kas-Sharir, for hours of fascinating and enriching meetings and fruitful collaborative work. I especially want to thank Naama, who hosted me for an internship at VMWare. I enjoyed every meeting we had at the campus, but beyond that, she invited me to her lovely apartment, and more than all made me feel at home during the internship.

I thank my candidacy and final exam committees—Yehuda Afek, Keren Censor-Hillel, Roy Friedman, Maurice Herlihy and Idit Keidar—for their time and insights.

And finally, thank you to Orr, my beloved, and to my marvelous family.

The generous financial help of the Technion, the Jacobs Fellowship, the Kenneth and Gloria Levy Graduate Fellowship and the Israel Science Foundation is gratefully acknowledged.

Contents

List of Figures

Abstract

Over the past two decades, the development of multicore processors has been a key avenue for enhancing computer performance. These processors enable multiple processes to execute concurrently, thereby boosting overall performance. To fully exploit this hardware, new algorithms are required to synchronize data access across multiple cores. A central focus of this thesis is on concurrent data structures, a fundamental building block of concurrent algorithms. This thesis also focuses on distributed systems, which incorporate multiple computers that communicate via message-passing across network channels and cooperate to perform shared tasks. Contemporary computing infrastructure relies on these systems since modern applications often demand computing power and reliability that a single computer cannot offer. Beyond high performance, concurrent and distributed systems must ensure correctness and durability in the face of failures. We explore fundamental concurrent and distributed algorithms that provide these essential properties.

We start this journey in the setting of concurrent data structures, where we study their size property. We present the first methodology for supporting a fast and correct size operation to a wide group of concurrent data structures. Next, we design efficient durable concurrent FIFO queues for computers with non-volatile memory, a new type of memory employing new semiconductor technologies developed by Intel and other companies. Our queues do not only demonstrate high performance, but are also provably optimal in minimizing interaction with the memory, and adhere to a general guideline we present for an efficient design of algorithms for non-volatile memory. When analyzing the correctness of concurrent data structures, linearizability is the most frequently used criterion. In another study, we point out a typo in the definition presented in its original paper and provide an amendment to make the definition complete.

Within the context of distributed systems, we focus on the fundamental challenge of fault tolerance, and consider asynchronous networks with unbounded channel noise which may completely corrupt all messages on all channels, thus requiring content-oblivious communication. We show how 2-edge connected networks may surprisingly tolerate unbounded channel noise, while any other network cannot. Finally, we explore distributed systems combining concurrency, namely, comprising multiple nodes where each node is a multicore machine. We formalize properties crucial to fast and robust distributed systems, and show inherent tradeoffs between them and properties desirable for multicore efficiency. This serves as a guideline in designing future combined systems with improved performance.

Notation and Abbreviations

BST	binary search tree	42
CAS	compare-and-swap	13
CLFLUSH	Flush Cache Line	98
CLFLUSHOPT	Flush Cache Line Optimized	98
CLWB	Cache Line Write Back	96
DAP	disjoint-access parallelism	71
DDAP	distributed disjoint-access parallelism	79
DRAM	dynamic RAM	9
DurableMSQ	durable MSQ	126
FIDS	Fast decision, Invisible reads, distributed Disjoint-access parallelism, and Serializability	67
FIFO	first in first out	6
GST	global stabilization time	15
HDDs	hard disk drives	9
MOVNTI	Store Doubleword Using Non-Temporal Hint	99
MSQ	Michael-Scott queue	96
NICs	network interface controllers	8
NVRAM	non-volatile RAM	9
ONLL	Order Now, Linearize Later	100
PDTSs	parallel distributed transactional systems	66
R-FIDS	Robust Fast decision, Invisible reads, Disjoint-access parallelism, and Serializability	67
RDMA	remote direct memory access	8
SFENCE	Store Fence	96
SizeBST	size-supporting binary search tree	43
SSDs	solid-state drives	9

Chapter 1

Introduction

Since the invention of the computer, one of the main goals of the computer industry has been to improve computer performance, turning it into an essential tool in modern society. At first computers contained a single core in which programs were executed, and continuous hardware upgrades of the core led to newer computers that ran the same software faster. However, in the early 2000s, further improvements using the same methodology were not possible anymore due to physical limitations (specifically, overheating). As a substitute, a main way to improve performance of computer processors in the last two decades has been producing processors with multiple cores, on which tasks may execute concurrently.

This, in turn, required new algorithms for programs that run on multiple cores, which will correctly synchronize their accesses to data in shared memory. The design of such concurrent algorithms to exploit multicore computers is a major challenge in modern computer science. There has been substantial work on the design of efficient correct concurrent data structures, which are fundamental building blocks of concurrent programming [e.g., MS96; Har01; Fra04; EFRvB10; BP12; MA13; NM14; YM16; BCP16; SKL+18; SGP18], in order to benefit concurrent algorithms at large. A significant part of this thesis focuses on this area.

However, a single computer is inadequate for modern applications. In our interconnected world, high-speed networks link computers, making nearly everything accessible via the Internet reliant on multiple machines. *Distributed systems* leverage multiple computers communicating through message-passing across a network of channels to perform mutual tasks. These systems are pivotal in contemporary computing infrastructure, bolstering large-scale applications by enhancing performance and reliability. Specifically, they boost system efficiency by improving its scalability (i.e., utilizing multiple servers to increase the overall system's throughput) and ensure continuous client service even amidst server failures or network disruptions. This thesis explores the area of distributed systems, and also addresses combined systems, made of multiple nodes where each node is a multicore machine.

Multiprocess systems, whether concurrent, distributed, or a combination of both (systems with multiple computers, each running multiple processes), involve multiple entities operating simultaneously, requiring correct and efficient synchronization and cooperation to achieve their mutual goal. In the realm of multiprocess systems, we have explored the limits

of computation to determine what is and is not achievable. This thesis outlines the space of possible solutions for various problems, characterizes what cannot be solved, and formulates guidelines within the feasible domain for efficient future implementations. Additionally, we push the boundaries of what is possible and introduce novel algorithms that improve upon previous literature, as we prove both theoretically and by measurements.

This thesis focuses on several elementary desirable properties of multiprocess systems, striving to achieve them all. One of them is maintaining correctness, namely, guaranteeing that their executions ”make sense”, meaning they are equivalent to legal sequential executions. Linearizability is the standard correctness property for concurrent implementations. In our research on concurrent algorithms, we have worked to achieve linearizability, and also identified a typo in its definition and corrected it.

As a fundamental goal of multiprocess systems is to enhance efficiency, mere correctness is insufficient if performance is poor, since a system that is correct but slow is of little use in practice. Thus, alongside correctness, we have focused on optimizing performance. In particular, our research includes improving performance tailored to specific hardware, such as high-performance first in first out (FIFO) queues for non-volatile main memory and efficient concurrent distributed transactional systems for modern high-bandwidth networks.

Correctness and performance are challenging both to define and to achieve, and our work has extensively addressed both these aspects. In addition to consistency and efficiency, fault tolerance is a critical aspect of multiprocess systems. Real-world systems inevitably face node failures and network issues, necessitating the design of algorithms that can handle such faults. Concurrent and distributed systems should maintain functionality despite these failures while preserving benefits like scalability and high availability. Our work addresses different fault types and strategies to mitigate them.

Our research detailed in this thesis has focused on ensuring correctness, optimizing performance, and enhancing fault tolerance in multiprocess systems. By addressing these key properties, we strive to develop robust systems that can operate effectively in real-world environments.

1.1 Correctness

In the realm of concurrent data structure algorithms, a vital foundational requirement is the ability to determine if a particular algorithm is correct. This is a simple task for non-concurrent data structures; for example, it is easy to determine for a FIFO queue whether an operation that removes an item from the queue returns a correct return value, which is the item in the head of the queue. However, in executions of concurrent data structures, several processes may perform operations on the data structure simultaneously, and the overlapping intervals of the operations make their desirable results unclear. For instance, if multiple processes concurrently add items to a concurrent FIFO queue, then it is not obvious which inserted item should be considered the first in the queue, thus, it is unclear what item a subsequent remove operation should return.

Correctness criteria for concurrent data structures are introduced to solve these unclarities and define what is legal in concurrent environments. *Linearizability* [HW90] is a widely accepted criterion for determining whether a concurrent execution is correct. It is the de facto correctness condition for concurrent data structures, widely used in theory and practice. Loosely speaking, linearizability classifies concurrent executions as correct if operations on shared objects appear to take effect instantaneously during the operation execution time. We have worked to achieve linearizability in our concurrent algorithms, and also further explored the linearizability definition itself. We found a typo in the definition which we explain and suggest a fix for.

1.1.1 Correct and efficient concurrent size

We bring an example of a very important problem which did not have a solution both correct and efficient until we designed one. The problem concerns computing the size of a data structure (i.e., the number of elements in it), which is a widely used property of a data set. However, for concurrent programs, obtaining a correct size efficiently is non-trivial. In fact, the literature does not offer a mechanism to obtain a correct size of a concurrent data set without resorting to inefficient solutions, such as taking a full snapshot of the data structure to count the elements, or acquiring one global lock in all update and size operations. To obtain the size efficiently, concurrent libraries like the concurrent library of Java return an estimate of the size and not the accurate value. We stretched the limits of existing size solutions and designed the first methodology for size which is correct and efficient, both theoretically and practically.

In Chapter 3 we present a methodology for adding a concurrent linearizable size operation to sets and dictionaries with a relatively low performance overhead. Our method incorporates metadata into the data structure, which the size operation uses to calculate its size. This metadata is updated atomically alongside the data structure modifications to keep it up-to-date. We choose appropriate metadata that enables the size as well as the other operations to efficiently operate on it. Theoretically, the proposed size operation has asymptotic complexity linear in the number of threads (independently of data-structure size). Practically, we evaluated the performance overhead by adding size to various concurrent data structures in Java—a skip list, a hash table and a tree. The proposed linearizable size operation executes faster by orders of magnitude compared to the existing option of taking a snapshot and counting elements in the snapshot, while incurring a throughput loss of $1\% - 20\%$ on the original data structure's operations.

1.1.2 Linearizability typo fix

A somewhat-neglected aspect of linearizability is restrictions on how pending invocations are handled, an issue that has become increasingly important for software running on systems with non-volatile main memory. Interestingly, the original published definition of linearizability includes a typo (a symbol is missing a prime) that concerns exactly this issue. In

Chapter 4, we point out the typo and provide an amendment to make the definition complete. We believe that pointing this typo out rigorously and proposing a fix is important and timely.

1.2 Performance

One of the main goals of multiprocess systems is to improve the system efficiency. The methodology for adding a concurrent efficient size operation described in Section 1.1.1 exemplifies our efforts to enhance performance. Another example is the efficient queues for non-volatile memory we discuss below in Section 1.3.1, which demonstrate our work on high performance targeted at a certain hardware. Next (in Section 1.2.1), we elaborate on another work focused on improved performance inspired by hardware advances.

1.2.1 Performance in Parallel Distributed Transactional Systems

Chapter 5 studies distributed transactional systems leveraging node parallelism, making them suitable for networks with enhanced throughput. Transactions are operation sequences atomically performed on a database. Distributed transactional systems run transactions employing multiple nodes, while parallel (or *concurrent*) transactional systems run transactions using multiple cores in a node. Traditionally, distributed and parallel transactional systems have been studied in isolation (see for instance the works [CDE+12; ZXS+21; LM10] on distributed transactional systems and [AHM11; AH12; AF15; BHG86; BDFG14; Pap79; PPR+15; AS08] on parallel ones), as they targeted different applications and experienced different bottlenecks. Distributed transactional systems did not incorporate multiple processes per node since in the past, the network was the main bottleneck—network throughput was relatively low, making a single process in each node sufficient to handle the message delivery rate. However, with high-bandwidth network links, multicore network interface controllers (NICs), remote direct memory access (RDMA) and kernel bypassing all contributing to increased network throughput, sequential processing within each node becomes a bottleneck and is no longer enough to handle the high throughput. Hence, distributed transactional systems must make use of the parallelism available on each server that they use. This led us to study systems that are both distributed and parallel.

We study the performance of these combined systems and show that there are inherent tradeoffs between a system's ability to have fast and robust distributed communication and its ability to scale to multiple cores. We start with formalizing properties of distributed transactional systems aimed at improved performance. These properties have all appeared in various forms intuitively in the literature [ZSS+15; KPF+13; SWL+20] and apply to many existing systems, yet they have never been formalized until now, as formalizing such properties is a delicate task and reaching a simple definition that also applies to a wide variety of applications is not easy. We then present impossibility results, demonstrating that it is impossible to achieve these properties while also attaining desirable multicore performance properties. There is an inherent tension between the natural distributed properties we present and well-known mul-

ticore performance properties in transactional systems, making it impossible for any system to satisfy all of these properties simultaneously. This serves as a guideline advising what is not achievable when designing transactional systems. Finally, we show positive results; it is possible to construct a parallel distributed transactional system if any one of the properties we study is removed.

1.3 Fault Tolerance

Fault tolerance is a system's ability to continue operating in the presence of faults and still meet its specification even when individual hardware or software components fail. Its importance stems from the fact that building a flawless system is practically unattainable. Various faults can occur in multiprocess systems; we specifically focus on node crashes and channel noise.

In the setting of distributed transactional systems described above (in Section 1.2.1), we address challenges posed by node crashes in distributed systems, examining how the system can overcome these failures and continue functioning with the remaining servers. Identifying failures poses a significant difficulty for asynchronous distributed systems, as it can be impossible to distinguish between a node crash and a network delay. Despite this, systems must still provide guarantees even in the event of crashes occur. Specifically in distributed transactional systems, we aim to complete transactions within a few round trips even if some servers crash, a property we formalize in our work (see *seamless fault tolerance* in Chapter 5). In the setting of distributed transactional systems described above (in Section 1.2.1), we address challenges posed by node crashes in distributed systems, examining how the system can overcome these failures and continue functioning with the remaining servers. Identifying failures poses a significant difficulty for asynchronous distributed systems, as it can be impossible to distinguish between a node crash and a network delay. Despite this, systems must still provide guarantees even in the event of crashes. Specifically in distributed transactional systems, we aim to complete transactions within a few round trips even if some servers crash, a property we formalize in our work (see *seamless fault tolerance* in Chapter 5).

In other works, detailed below, we investigate other types of faults: crash failures of multicore systems with non-volatile RAM (NVRAM), and channel noise in distributed systems.

1.3.1 Durable Concurrent Queues

We consider multicore systems with NVRAM, such as the Intel Optane memory architecture [Int19]. NVRAM is a new memory technology that provides persistence, namely, preserving data after a power loss. It offers large storage capacity similar to hard disk drives (HDDs) and solid-state drives (SSDs), while also providing rapid access and byte-addressability—features that make it comparable to dynamic RAM (DRAM). Hence, NVRAM facilitates the development of persistent data structures that are significantly faster than those relying on HDDs and SSDs, as extensively explored in recent years [e.g., CJ15; LLS+17; OLN+16; YWC+15; FHMP18;

DDGZ18; ZFS+19; MIS20; CCA+11; CFR18; KPS+16; MMT+18; RCFC19; tea; VTS11; WRL19; ZZLS19; IMS16; FBW+20; FPR21]. In these data structures, data retained after a power loss is utilized during system recovery following a crash. However, caution is necessary because cache and register contents are lost during a crash, potentially leaving the data in main memory incomplete. Special attention is required when recovering a concurrent algorithm due to interactions between different threads and their dependencies. Persist instructions (i.e., flushes and fences) may be used to ensure that certain data reaches the memory before proceeding to the next step of the program. For *lock-free* concurrent data structures, it suffices to persist any memory location right after accessing it [IMS16]. This would result in a correct implementation, but a highly inefficient one since persist instructions are extremely expensive.

A known guideline for making data structures durable while incurring low overhead advises incorporating the fewest possible number of blocking persist operations [CGZ18]. In Chapter 6 we show that focusing on minimizing the number of persist instructions is important, but not enough. We find that access to flushed content is of high cost due to cache invalidation in existing architectures. We accordingly formulate a guideline for future efficient durable algorithms for NVRAM, dictating they should reduce access to recently flushed cache lines. We target concurrent FIFO queues as an example, and given this finding, we present durable queues that minimize blocking persist operations as well as access to flushed content. Theoretically, our design is optimal in the sense that it incurs a single blocking persist instruction per update operation (which is the minimum possible) and performs no access to flushed data. We also point out that any object with a deterministic sequential specification can be implemented with a single blocking persist instruction per update operation and no access to flushed data. This may guide algorithm designers in tailoring efficient implementations for durable data structures. Practically, our evaluation shows that the proposed design outperforms state-of-the-art durable queues, demonstrating the importance of reduced accesses to flushed content.

1.3.2 Fully-Defective Distributed Networks

Another type of fault that we investigate is channel noise corrupting messages in asynchronous distributed systems. Some assumptions restricting the noise are necessary; unlimited noise could prevent any meaningful communication. Previous work has limited either the number of noisy channels [Dol82; SW90; Pel92; SAA95; HP21a; HP21b] or the total amount of corruption [HS16; CGH19; GKR19; ADHS20].

We focus in Chapter 7 on alteration noise, where message content can be changed but messages cannot be deleted or injected, without bounding the number of noisy channels or the amount of noise. Equivalently, the network could be envisioned as one where nodes communicate by sending pulses to their neighbors, which could be the case if the nodes possess very basic communication hardware. This model of unbounded alteration noise, which may corrupt the content of all messages, might seem too harsh for any reliable communication.

However, we show that this is not the case, and provide a full characterization identifying which network topologies enable non-trivial computations in this model and on which network topologies non-trivial computations are impossible (by *non trivial* we refer to computations that give output or terminate).

The graph family that enables non-trivial computations under unbounded alteration noise is 2-edge-connected graphs (i.e., graphs that remain connected upon the removal of any edge). The key structural property of these graphs that we leverage is the existence of a directed (non-simple) cycle that goes through all nodes where each edge that appears multiple times in the cycle has the same direction in all its occurrences, as shown by Robbins [Rob39]. The core of our technical contribution is presenting a construction of such a Robbins cycle in fully-defective networks, and showing how to communicate over it despite total message corruption. These are obtained in a *content-oblivious* manner, since nodes must ignore the content of received messages.

Chapter 2

Preliminaries

Concurrent System Model

We consider a standard shared memory setting [DHW97], with a set of asynchronous threads communicating by accessing shared memory using the atomic primitives read, write and read-modify-write, specifically $\mathrm{compare\text{-}and\text{-}swap}$ (CAS). A CAS instruction on an object takes an expected value and a new value. It atomically obtains the object's current value and swaps it with the new value if the current one equals the expected value. The return value indicates whether the substitution was performed: its compareAndSet variant returns a corresponding boolean value; its compareAndExchange variant returns the obtained current value. If the substitution was performed we say that the CAS is successful. A single-word CAS is supported on nearly all platforms (possibly using the equivalent LL/SC instructions). Some platforms also support a double-width CAS, which applies to data residing on two adjacent words.

Concurrent Executions

We follow the terminology of the paper on linearizability [HW90]. An execution of a concurrent system is modeled by a history. A *history* is a finite sequence of operation *invocation* and *response* events. Each invocation or response event is associated with some object and some process. An invocation includes also an operation name and argument values, and a response includes a termination condition and results. A response *matches* an invocation if it is associated with the same object and process. An invocation is *pending* in a history if no matching response follows the invocation. An *extension* of H is a history constructed by appending to the end of H responses to zero or more pending invocations of H. A *subhistory* of a history H is a subsequence of the events of H. *complete(H)* is the maximal subhistory of H consisting only of invocations and matching responses, without any pending invocations. For a process P, the *process subhistory* $H|P$ is the subsequence of all events in H associated with the process P. For an object x, the *object subhistory* $H|x$ is the subsequence of all events in H associated with the object x. Two histories H and H' are *equivalent* if for every process P, $H|P = H'|P$.

A history H is *sequential* if it comprises a sequence of pairs of an invocation and a matching response, except possibly the last invocation, which might be the last event in the history, not accompanied by a matching response. A history that is not sequential is *concurrent*. A history is *well-formed* if each of its process subhistories is sequential. A *single-object* history is one in which all events are associated with the same object. A *sequential specification* for an object is a prefix-closed set of single-object sequential histories for that object. A sequential history H is *legal* if each object subhistory $H|x$ belongs to the sequential specification for x.

An *operation* in a history is a pair consisting of an invocation and the next matching response. An operation e_0 *precedes* (synonymously *happens before*) an operation e_1 in a history H if e_0 ends before e_1 begins, namely, e_1's invocation event occurs after e_0's response event in H. Precedence in H induces a partial order on operations of H, denoted $<_H$. Informally, $<_H$ captures the "real-time" precedence order of operations in H. We stress that only invocations that have matching responses are considered *operations* and the order $<_H$ applies only to them.

Linearizability

A history is considered *linearizable* [HW90; SHP21a] if each completed (i.e., non-pending) operation appears to take effect at once, between its invocation and its response events, in a way that satisfies the sequential specification of the objects. Each pending operation is required to either take effect at once after its invocation in a way that satisfies the sequential specification of the objects, or not take effect at all. The point in time in which an operation takes effect is denoted its *linearization point*. A concurrent data-structure is *linearizable* if all its executions are linearizable.

Progress Properties

A concurrent object implementation is *lock-free* [Her91] if each time a thread executes an operation on the object, some thread (not necessarily the same one) completes an operation on the object within a finite number of steps. Thus, lock-freedom guarantees system-wide progress.

A concurrent object implementation is *wait-free* [Her91] if any thread can complete any operation in a finite number of steps, regardless of the execution speeds of other threads. Thus, wait-freedom guarantees individual thread progress.

Distributed System Model

To explore fully-defective networks, we consider a standard asynchronous message-passing system [see, e.g., AW04] in which the network is modeled as an undirected graph $G = (V, E)$, where the nodes V represent computing devices and the edges E are bi-directional communication channels between them. To conduct a computation the nodes run a protocol, which is an asynchronous event-driven distributed algorithm. Initially, each node begins with a private input and generates messages to send to zero or more of its neighbors according to the

protocol. Afterwards, the protocol behaves in an event-driven manner, and upon receiving a message, a node performs some local computation and produces messages designated to zero or more of its neighbors. Communicating a message over some link of the network takes arbitrary positive finite time. In the fully-defective network model we consider, alteration noise can corrupt the content of any message communicated over any channel. However, the noise cannot completely delete a message nor can it inject a message on a link in which no message was sent.

Parallel Distributed System Model

To investigate parallel distributed transactional systems, we formalize a model that combines both shared memory and message passing systems. We consider a message-passing model that includes server nodes and client processes. Each server node runs multiple processes, such that node processes within a single node communicate with each other via shared memory. That is, they access shared base objects through primitive atomic operations, such as read, write, read-modify-write (compare-and-swap, test-and-set, fetch-and-increment, etc.). Messages are sent either between two nodes or between clients and nodes0. We consider partial synchrony [DLS88]; messages can be arbitrarily delayed until an a priori unknown *global stabilization time (GST)*, after which all messages reach their target within a known delay. Nodes can fail by crashing; if a node crashes then all processes on the node crash as well. We do not consider failures where individual processes crash and we assume clients do not fail. In this setting we assume message losses are handled by the networking layer, and do not consider them further. The protocols according to which processes run transactions, handle messages and access shared memory are detailed in Section 5.2 in Chapter 5.

Chapter 3

Concurrent Size

This chapter is based on the work presented at [SP22b] and [SP22a] (and the code is available at [SP22c]).

3.1 Introduction

A fundamental, widely used, property of a data structure is its size (i.e., the number of elements it contains). In Java, for example, any collection or map class that implements one of the elementary interfaces java.util.Collection or java.util.Map [22] must implement a size method. Interestingly, implementing an efficient and correct size operation for a concurrent data structure is non-trivial. For a formal treatment, we use linearizability as the correctness criterion of concurrent executions [HW90; SHP21a], but the discussion below also applies to other intuitive correctness criteria.

The literature does not offer an acceptable solution to implementing a correct size operation, and existing implementations give up correctness in order to avoid a significant performance deterioration. For example, the non-blocking collections and maps in the java.util.concurrent package [Lea04] implement a non-linearizable size method that returns an estimate of the size. The returned estimate may be inaccurate when the object is concurrently modified during the execution of size. In contrast, a linearizable size operation would tolerate concurrent update operations and retrieve the exact number of elements in the data structure at some point during the execution of the size operation.

Existing solutions are incorrect or inefficient. Ignoring concurrency, one can determine the size of a data structure simply by traversing it and counting the number of encountered items. This is the approach taken by the size method of Java's ConcurrentLinkedQueue and ConcurrentLinkedDeque. This approach is fine for a sequential execution, but for a concurrent execution this implementation is not linearizable. The following is a worst-case scenario for this implementation. Consider an execution on a linked list with the single item 1. Assume a thread T, running this size implementation, starts the traversal from the node containing 1 and then gets preempted. At this point, the following steps may occur repeatedly: some thread appends a node with the item 2 to the end of the list, increasing the list's size

to 2; next, T gets scheduled, resumes its traversal and proceeds to this new node; then, some thread deletes the node containing 1, so the list's size is 1 again. Next, some thread inserts 3 and deletes 2, letting T see a third element, etc.; until eventually—after some item s is appended—the thread T gets to the end of the list before another thread gets the chance to insert an additional item. In this scenario, T will erroneously return a possibly large s as the list size, while in practice the list size never exceeded 2. While this is a worst-case scenario, one can envision many other scenarios in which the returned value would be incorrect.

Alternatively, it is possible to obtain a correct size implementation by obtaining a linearizable snapshot of the data structure (e.g., using any of the methods in [WBB+21; PT13; NHP22; AB18]) and then iterate over the returned snapshot to count the number of elements in the snapshot. While correct, this solution is inefficient, yielding a time complexity linear in the number of elements in the data structure.

If we want to avoid such a high cost for the size operation, then we need to keep some metadata that allows computing the size of the data structure efficiently when needed, and let the data-structure operations maintain this metadata. Naively, the metadata would just be the current size value. A natural attempt to implement such a size operation would be to keep the size in a designated field of the data structure and let the operating threads update it with each operation that affects the size. An insert operation would execute a size increment after inserting its item, and a delete operation would execute a decrement of the size field after performing the deletion. Java's ConcurrentSkipListMap and ConcurrentHashMap use such an implementation. However, the separation between the data-structure update and the metadata update foils linearizability. As an example, consider n threads that are preempted exactly after inserting an element to the data structure and before updating the size field. At this point the size field would be off by n and thus inconsistent with a view of another thread that actually reads the data structure.

A simplified execution for one updating thread is depicted in Figure 3.1. There, only one update operation is ever executed on the data structure: one thread inserts 1 into an empty structure. Another thread that starts by calling contains(1), sees that the data structure already contains the single element in it, but then it calls size() and receives 0. We executed this simple program on Java's ConcurrentSkipListMap several times, and we actually witnessed executions that reproduced the contradicting result as depicted in Figure 3.1. This demonstrates that the size method of ConcurrentSkipListMap is not linearizable. The core issue is the separation between the actual data structure update and the subsequent update of the metadata.

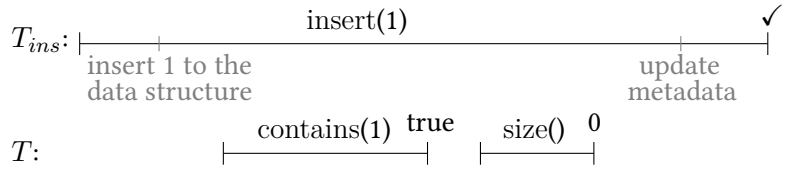


Figure 3.1: An execution with conflicting contains and size results due to the separation between updating the data structure and the size metadata

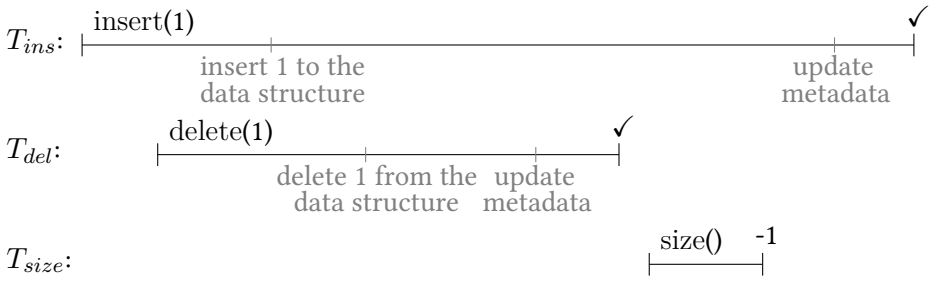


Figure 3.2: An execution that yields a negative size due to the separation between updating the data structure and the size metadata

Furthermore, updating the metadata separately from updating the data structure may yield a size execution that returns a negative number. This means that the size operation is not only non-linearizable, but it can also not satisfy any correctness criterion that requires method calls to appear to happen in a one-at-a-time sequential order, e.g., it is not quiescently consistent [AHS94; SZ96; HS08] nor sequentially consistent [Lam79; HS08]. Consider the following execution (depicted in Figure 3.2). Thread T_{ins} inserts an item to the data structure, and before it updates the metadata, thread T_{del} deletes that item and updates the metadata, registering its decrement. At this point, thread T_{size} calls size() that returns -1 based on the metadata, which currently reflects only the deletion and not yet the insertion. The separation between the data-structure and metadata updates results here in updating the metadata in a reversed order, which is impossible since the deletion cannot succeed if it happens before the insertion. The returned size exposes this impossible operation order. The method calls in this execution do not appear to happen in a one-at-a-time sequential order since size would never return a negative result in a legal sequential execution.

A more complex metadata maintenance is proposed by Afek et al. for computing the size more efficiently [AST12]. But they, too, update the metadata after the data-structure update, and so their implementation suffers from the same problems. (We elaborate on issues in [AST12] in Section 3.3.1.)

A third alternative for implementing the size operation is to use locks to prevent a size operation from exposing a temporary inconsistency between the data structure's state and the metadata. This too would create a severe bottleneck and deteriorate performance significantly.

In this work we propose an efficient linearizable size implementation. To the best of our knowledge, this is the first size solution that provides both linearizability and efficiency (namely, not iterating over all elements of the data structure or using coarse-grained locking). We present a methodology for adding a linearizable size operation to concurrent data structures that implement a set or a dictionary. Our methodology yields data structures that satisfy the following attractive theoretical properties:

1. The time complexity of the size operation is linear in the number of threads.
2. The size operation is wait-free, namely, a thread running a size operation completes the operation within a finite number of steps, regardless of the activity of other threads.
3. The (asymptotic) time complexity of the original data-structure operations is preserved.

4. The progress guarantees of the original data structure are preserved. Namely, wait-free methods of the original data structure remain wait-free in the transformed data structure, and the same goes for lock-free or obstruction-free methods.

To achieve Property (1), we keep always-consistent metadata, from which the size can be correctly computed. To prevent operations from exposing inconsistencies similar to the examples of Figures 3.1 and 3.2, we work hard to achieve a single linearization point in which the data structure is modified and the metadata gets updated simultaneously. This is obtained by letting an operation appear as completed to other operations only when the metadata update occurs. Formally, the update of the metadata becomes the single linearization point of the entire data structure operation. Dependent data-structure operations are adapted to comply with the new linearization point, and help completing concurrent operations when necessary. For instance, a $\mathrm{delete}(k)$ by thread T_2 that encounters an ongoing $\mathrm{delete}(k)$ by another thread T_1 which has already deleted the key from the data structure, must help T_1 to update the metadata in order to complete the obstructing $\mathrm{delete}(k)$ before returning a failure. It cannot block and wait for T_1 to update the metadata, since that might change the progress guarantees of the delete operation and foil Property (4).

Helping another operation implies updating the metadata on its behalf. As always with multiple threads helping to execute a single operation, care has to be taken for the operation to be executed only once. We keep per-thread counters as the size metadata, and use a corresponding mechanism that enables helpers to determine whether the metadata already reflects the helped operation, to prevent a wrong double update of the metadata on behalf of the same operation. This mechanism enables helpers to efficiently make a determination and update the metadata if necessary, thus achieving Property (3).

The size of a data structure is a fundamental property of a data set and having a methodology for obtaining an efficient accurate solution for it seems like an important point in the design space, which is currently missing in the literature. Using inaccurate solutions may yield unexpected results, e.g., sizes that the data structure never had and even a negative size. Such results may in turn yield unexpected bugs that may put the reliability of an entire system at risk. A reliable solution is especially desirable in dynamic programming languages that favor correctness over performance, such as Python and Ruby, which use a global interpreter lock in their reference implementations and are expected to behave reliably even in optimized implementations that shed the global interpreter lock to obtain parallelism. This follows the line of previous works [e.g., DTM+18; MRG16] that present solutions for reliable efficient parallelism.

In order to evaluate the performance overhead of the linearizable size operation, we added the size operation using the methodology described in this work to various concurrent data structures in Java: a skip list, a hash table and a tree [SP22c]. The proposed linearizable size operation executes faster by orders of magnitude compared to counting the elements of a linearizable snapshot. It also demonstrates scalability and insensitivity to the data-structure size. However, obtaining a linearizable size operation does come with some cost, incurring a

throughput loss of $1\% - 20\%$ on the original data structure's operations.

The rest of this chapter is organized as follows. Section 3.2 introduces some basic terminology. Section 3.3 surveys relevant work on snapshots. We then describe our methodology, starting with the transformation of a linearizable data structure into one that uses our size mechanism in Section 3.4, and proceeding with the size mechanism itself: the metadata design is covered in Section 3.5, and Section 3.6 describes how the size is obtained in a wait-free form. We describe possible optimizations to our methodology's implementation in Section 3.7. In Section 3.8 we argue about the properties the methodology satisfies. Section 3.9 presents an evaluation of the methodology applied to different data structures in a variety of workloads. We conclude in Section 3.10.

3.2 Terminology

A *set* is a collection of keys without duplicates, supplying the following interface operations: an $\mathrm{insert}(k)$ operation which inserts the key k if it does not exist or else returns a failure; a $\mathrm{delete}(k)$ operation which deletes k if it exists or else returns a failure; and a $\mathrm{contains}(k)$ operation which returns true if and only if k exists in the set.

A *dictionary* (synonymously *map* or *key-value map*) is a collection of distinct keys with associated values, with operations similar to the ones of a set but with values integrated in them. Throughout the chapter we will refer only to sets for brevity, but all our claims apply to dictionaries as well.

3.3 Related Work

A *snapshot object* [AAD+93] is an abstraction of shared memory made of an array of m cells, supporting two operations: $\mathrm{update}(i, v)$ that writes v to the i-th cell of the array, and $\mathrm{scan}()$ that returns the current values of all m locations (i.e., a snapshot of the array). The *atomic snapshot* problem is to implement such an object such that its two operations are linearizable and wait-free. Jayanti [Jay05] presents algorithms that solve the problem with optimal time complexity. We build on the fundamental ideas of Jayanti [Jay05] in this work to design a wait-free size operation.

However, this scheme is not suited for multiple concurrent scan operations and does not allow other operations (such as reading a specific cell) to occur concurrently. Petrank and Timnat [PT13] extend Jayanti's idea and introduce a technique for adding a linearizable wait-free snapshot operation to a concurrent set data structure. In supporting concurrent size operations, we use their method to support multiple concurrent snapshot operations.

An alternative approach by Wei, Ben-David, Blelloch, Fatourou, Ruppert, and Sun [WBB+21] and Nelson-Slivon, Hassan, and Palmieri [NHP22] obtains snapshots of concurrent data structures more efficiently, at the cost of higher space overhead. They keep copies of modified nodes and let the snapshot operation advance a timestamp. This timestamp is then used to read the content of the data structure at the time the snapshot was taken. To support such

a read of old values, operations on the data structure are responsible to maintaining lists of previous values of mutable fields. Specifically for obtaining the size, one may take a snapshot and use the returned timestamp to traverse the data structure at that time and count elements.

Literature on range queries may be also utilized to take a full snapshot of a data structure. For instance, Arbel-Raviv and Brown [AB18] propose to implement range queries using epoch-based memory reclamation.

The above snapshot algorithms can be used to obtain a linearizable size, but using them for this purpose is an overkill. The comparison we make in Section 3.9 to the algorithms of Petrank and Timnat [PT13] and Wei, Ben-David, Blelloch, Fatourou, Ruppert, and Sun [WBB+21] demonstrates the clear benefit of using our algorithm which is tailored for obtaining the size.

3.3.1 Inaccuracies of the algorithm in [AST12]

We showed in Section 3.1 that updating the data structure and the size-related metadata separately, as is done in the algorithm for data-structure size presented in [AST12], makes the algorithm non-linearizable and also not satisfying the weakest correctness principle defined in [HS08] which requires method calls to appear to happen in a one-at-a-time sequential order. But even if update operations somehow update the data structure and the metadata atomically, the algorithm of Afek, Shavit, and Tzafrir [AST12] will still not satisfy the above-mentioned correctness principle, due to another issue we elaborate on next. We start with a demonstrating execution, which produces an impossible negative size, hence, no reordering of its method calls forms a legal sequential execution.

Consider an execution with 3 threads executing their operations concurrently: thread T_0 executes an insertion of an item to the data structure, thread T_1 executes a deletion of the same item from the data structure, and thread T_2 executes a size call. First, T_0 and T_1 start executing their operations. T_0 inserts the item to the data structure and then T_1 successfully removes it. After operating on the data structure, they both call the algorithm's $wait\_free\_update$ method to update their values in the array g_mem. During these method calls, they obtain g_seq when its value is 0, and before they proceed to updating g_mem, T_2 starts its size execution. It performs $scan\_seq := FAI(g\_seq)$ which results in $scan\_seq = 1$, and later starts collecting g_mem's values. It obtains $[0, 0]$ from $g\_mem[0].\_recent$ and adds 0 to $size$. At this point, T_0 resumes its execution, and writes $[1, 0]$ to $g\_mem[0].\_recent$ (this value was already missed by T_2). Then T_1 resumes its execution, and writes $[-1, 0]$ to $g\_mem[1].\_recent$. Now T_2 continues scanning the array. It obtains $[-1, 0]$ from $g\_mem[1].\_recent$ and accordingly adds -1 to size, and then $[0, 0]$ from $g\_mem[2].\_recent$ and adds 0 to size. Subsequently, it returns the incorrect size -1.

To analyze how this happened, we examine the linearization points of the operations on the array g_mem. The linearization point of the size operation by T_2 is when it increments g_seq using FAI; the linearization point of the insertion by T_0 is when it writes to $g\_mem[0].\_recent$. The problem stems from the linearization point of the deletion by T_1. It cannot be placed (like erroneously mentioned in [AST12]) when T_1 writes to $g\_mem[1].\_recent$,

because it must occur before the linearization point of the size operation that observed the deletion. Instead, it is placed in retrospect right before the linearization point of the size operation. This linearization scheme of possibly placing in retrospect linearization points of updates that the size operation observes in its scan long before they write their value, is adopted from the single-scanner algorithm of Riany et al. [RST01], on which the size algorithm in [AST12] is based. The scheme was intended for the atomic snapshot problem, where there are no dependencies between the update operations. However, when handling dependent data-structure operations, they cannot be freely reordered when the size observes them in its scan. In the execution described above, reversing the order of an insertion and a following deletion of the same item is unacceptable, since the deletion cannot succeed if it happens before the insertion, and thus cannot legally decrement the size before the insertion increments it. In conclusion, the correctness problem of the suggested size algorithm stems from the linearization order the size operation dictates: if a size operation S observes during its scan a value, written after S's linearization point by a delayed update U, it dictates in retrospect to place U's linearization point before S's linearization point – which might occur before linearization points of update operations that U depends on.

3.4 Data-Structure Transformation

In this section we specify how the fields and methods of a linearizable data structure can be modified in order to transform it into a data structure that uses our size mechanism. To efficiently obtain a linearizable size, we keep metadata from which the size may be computed. But unlike previous work, we make the data structure and the metadata change (linearize) simultaneously. The data-structure operations are responsible to maintain the metadata. The main idea is to make sure that updates are not visible to other operations until their metadata is updated. The way to enforce that, is to let each operation complete work for previous related operations, so that it does not view any intermediate states. The details follow.

Successful operations update metadata The first modification is to let each successful insert or delete operation (i.e., an operation that succeeds to insert a new key or delete an existing key respectively) update the metadata to reflect the operation's effect on the size.

Operations help concurrent operations on the same key update metadata To prevent operations from exposing inconsistencies similar to the examples of Figures 3.1 and 3.2, we linearize data-structure operations that alter the size at the time the metadata is updated to reflect them (informally, linearizing means logically considering them as applied). Dependent data-structure operations are adapted to comply with the new linearization point: if they observe that successful insert or delete operations that they depend on have accomplished their original linearization point, they help them update the metadata so that they reach their new linearization point. For example, a $\mathrm{contains}(k)$ that encounters a node with the key

k inserted by a concurrent insert(k) cannot return true before ensuring that the insert is reflected in the metadata.

We focus on data structures implementing a set (i.e., a collection of distinct keys) or a dictionary (i.e., a collection of distinct keys with associated values) that provide standard insert, delete and contains operations. In such data structures, an operation on some key logically depends only on operations on the same key. Accordingly, when an operation on some key encounters a node with that key, it acts as follows: if the node is unmarked, it updates the metadata on behalf of the insert operation that inserted the node, to guarantee it is complete (in case the metadata is not yet updated with this insert); if the node is marked as deleted, it ensures the metadata reflects the delete operation that marked the node before proceeding with its own execution.

Successful operations leave a trace for helpers For operations to help unfinished operations on the same key to update the metadata, they must observe these unfinished operations. To facilitate this, successful insert and delete operations prepare an UpdateInfo object with the information required by helpers for updating the metadata on their behalf, and reference it from the node on which they operate. An insert creates an UpdateInfo object and places a reference to it in the node it is about to link, in a new insertInfo field we add to node objects. A delete also creates an UpdateInfo object, and needs to reference it from the node it deletes. To this end, we rely on a deletion pattern introduced by Harris [Har01] and commonly used in concurrent data structures [e.g., Har01; HHL+05; HLLS07; Fra04; ST05], where a node is first marked as deleted and then physically unlinked. We install the delete information together with the marking, as demonstrated in the following examples.

When the original marking step already marks the node as deleted by installing an object with information about the delete operation (this is true, for instance, for the binary search tree of Ellen, Fatourou, Ruppert, and van Breugel [EFRvB10]), then a deleteInfo field referencing the delete's UpdateInfo object may be simply placed inside that object. When the original marking step nullifies the node's value field (as in Java's ConcurrentSkipListMap), in the transformed data structure instead of setting the value field to NULL, it may be set to a reference to the UpdateInfo object. When the original marking step sets a bit in the node's next field (like in Harris's linked list [Har01]), a new deleteInfo field in the node may be set to reference the UpdateInfo object before the marking step.

Metadata is updated before unlinking a marked node The metadata must be updated on behalf of a delete before the relevant node is unlinked. To see why, assume the metadata is updated to reflect a delete(k) only after it completes to operate on the data structure, including unlinking the node with the key k. In this circumstance, a dependent operation like contains(k) might run in between, and then it will not observe the relevant node and will thus not assist the delete operation update the metadata. Such a contains(k) would return false though delete(k) has not yet updated the metadata, hence, is not yet linearized. Therefore, the metadata is updated before any unlinking attempt: the delete(k) operation itself updates

the metadata after marking the node and before unlinking it; and any other operation that attempts to help unlinking the marked node is also required to update the metadata on behalf of delete(k) beforehand.

Adding size functionality An instance of a SizeCalculator object (described in detail in Section 3.6.1), responsible for keeping the metadata and computing the size, is referenced from the transformed data structure, and a size method that uses it to retrieve the size is added to the data structure.

3.4.1 Specific Examples and the SizeCalculator Object

Figure 3.3 demonstrates how the transformation described above may be applied to standard linearizable linked list, skip list and hash table that implement a set. A similar transformation with minor adaptations will apply to implementations of a dictionary. The transformation may also be applied to search trees with some adaptations.

At the core of our size mechanism stands the SizeCalculator object. We elaborate on this object, responsible for the size calculation, in Section 3.6.1. For now we just need to be familiar with its interface methods: updateMetadata is called with an UpdateInfo object associated with an insert or delete operation for updating the metadata stored in the SizeCalculator to reflect that operation. This method may be called by both the operation initiator and helpers. We explain in Section 3.5 how SizeCalculator prevents double update of the metadata on behalf of the same operation. createUpdateInfo is called by insert and delete operations to produce an object that will be published to helpers, with the information required for updating the metadata on their behalf. compute is the method used to retrieve the size of the data structure efficiently (using the metadata).

A SizeCalculator reference field named sizeCalculator is placed in the data structure, and initialized to hold a SizeCalculator instance. Its methods are called in the appropriate places in the data-structure operations, as can be seen in Figure 3.3. In addition, an insertInfo field referencing an UpdateInfo object is placed in the data structure's node objects. A similar deleteInfo field is placed in the appropriate place, as described above. Since the UpdateInfo record contains the information required for updating the metadata to reflect the associated operation, its content is coupled with the size metadata, so its description is deferred to Section 3.5.

3.4.2 Applicability

We focus on a transformation for linearizable data structures that implement the highly prevalent set or dictionary data types. However, the presented ideas may be adapted to other data types. Our transformation recipe requires that the delete operation of the original data structure be linearized at a marking step and not at an unlinking step, to ensure consistency with the size metadata. Otherwise, if operations on k that encounter a marked node with the key k ignore the mark, and consider k as deleted only when its node is unlinked, that might be

```
1  INSERT = 0, DELETE = 1
2  class TransformedDataStructure:
3      TransformedDataStructure():
4          initialize as originally
5          sizeCalculator = new SizeCalculator()
6      contains(k):
7          search* for a node with k as originally
8          if not found: return false
9          else if found unmarked node:
10             sizeCalculator.updateMetadata(node.insertInfo, INSERT)
11             return true
12         else: // found marked node
13             sizeCalculator.updateMetadata(node's deleteInfo, DELETE)
14             return false
15     insert(k):
16         search* for the place to insert k as originally
17         if k is already present in an unmarked node:
18             sizeCalculator.updateMetadata(node.insertInfo, INSERT)
19             return failure
20         if k is present in a marked node:
21             sizeCalculator.updateMetadata(node's deleteInfo, DELETE)
22         insertInfo = sizeCalculator.createUpdateInfo(INSERT)
23         allocate newNode as originally with k and the other relevant data, and additionally with
               insertInfo
24         insert newNode as originally (in case of failure proceed as originally)
25         sizeCalculator.updateMetadata(insertInfo, INSERT)
26         return success
27     delete(k):
28         search* for a node with k as originally
29         if not found: return failure
30         if found a marked node:
31             sizeCalculator.updateMetadata(node's deleteInfo, DELETE)
32             return failure
33         sizeCalculator.updateMetadata(node.insertInfo, INSERT)
34         deleteInfo = sizeCalculator.createUpdateInfo(DELETE)
35         mark node with deleteInfo (in case of failure proceed as originally)
36         sizeCalculator.updateMetadata(deleteInfo, DELETE)
37         unlink node
38         return success
39     size():
40         return sizeCalculator.compute()
41 *For each encountered marked node along the search, in case of unlinking it in the original algorithm,
      call sizeCalculator.updateMetadata(node's deleteInfo, DELETE) before unlinking it.
```

Figure 3.3: A transformed data structure

inconsistent with the size metadata which is updated to reflect the deletion before the unlinking. Instead, by our requirement, operations on k in the original data structure consider the node as removed when it is marked, and in the transformed data structure they help update the metadata on behalf of the $\mathrm{delete}(k)$ that marked the node and only then treat the key k as deleted.

In case of a data structure that linearizes the delete operation at an unlinking step and not in the prior marking step, it is usually not difficult to adjust it to have the marking as the linearization point of delete. We made this adjustment to the binary search tree of Ellen, Fatourou, Ruppert, and van Breugel [EFRvB10] in order to apply the transformation to it and evaluate its performance.

3.5 The Size Metadata

In our transformation, operations may help other operations update the metadata. Hence, we must prevent a double update of the metadata on behalf of the same operation. We use metadata which enables threads that operate on the data structure to determine whether the metadata already reflects a certain operation, and update it otherwise. The $\mathrm{SizeCalculator}$ object holds the array $\mathrm{metadataCounters}$ as the metadata, containing two counters per thread: an insertion counter and a deletion counter, which indicate the number of successful insertions and deletions the thread has performed so far on the data structure. Separating the insertion from the deletion counter allows determining whether an insert (or a delete) operation has been reflected in the counters. If an insert follows a delete, a single counter (incremented on each insertion and decremented on each deletion) cannot indicate if the two operations completed or none of them. Two separate counters allow a simple concise indication of which one of the two operations is reflected in the counters. Next we describe how insertions are handled; deletions are handled similarly.

The per-thread monotonic insertion counters enable to immediately detect whether a certain insert operation by a certain thread is reflected in the metadata, and otherwise ensure that it is reflected via a single CAS: When $\mathrm{updateMetadata}$ is called on behalf of a thread T's i-th successful insert operation by either T or helpers, if T's insertion counter is $\geq i$, it leaves the counter as is since the operation is already reflected in the metadata; else, it uses a CAS to increment it from $i-1$ to i. There is no need to repeat the CAS in case of failure, since that might happen only when another thread succeeds to perform the same update.

To help another operation update the metadata, a helper needs to know on which counter in $\mathrm{metadataCounters}$ it should operate and its target value. This dictates the information that the i-th insert operation by thread T leaves for helpers in an $\mathrm{UpdateInfo}$ object: T's thread ID, which will be used as an index to the $\mathrm{metadataCounters}$ array, and i, which is the counter's target value (which is simply the current value of T's insertion counter in $\mathrm{metadataCounters}$ plus 1).

The size may be calculated from $\mathrm{metadataCounters}$ as the difference between the sum of insertion counters and the sum of deletion counters. But naively reading the values one by

one may result in an inconsistent (non-linearizable) size, because we may obtain a collection of values that never existed simultaneously in the array. We need to obtain a snapshot of values the array counters had at some point in time, but we cannot use locks to achieve this atomicity as we aim for a wait-free size. Next we explain how we manage to achieve that.

3.6 Mechanism for Wait-Free Size

The size operation needs to obtain a linearizable snapshot of the metadataCounters array, from which it will be able to compute a consistent size. As the size of this array is twice the number of threads, our solution is the most beneficial (in comparison to computing the size by iterating over a snapshot of the data structure) for applications that usually use data structures with much more elements than the number of threads. If size naively read metadataCounters cell by cell, it could obtain an inconsistent view of the array. For example, consider an execution in which a size operation starts scanning the array, but after it reads the insertion counter of some thread T, this thread inserts an item and then removes it. Now both T's insertion and deletion counters equal 1, and when the size operation resumes it reads the new value of T's deletion counter and returns -1 as the size. The problem here is that the size operation captured the delete's update of the array but missed the preceding insert's update.

To overcome this problem and obtain a linearizable snapshot of the counters array in a wait-free form, we adopt the basic idea of [Jay05]'s single-scanner single-writer snapshot algorithm, which is as follows. After an update operation writes to the main array, it checks if a concurrent scan operation is in the process of collecting the main array's values. If so, the scan has maybe already read the relevant cell and missed the new value, thus the update forwards the new value from the main array to a designated second array. The scan operation begins with a collection phase to collect the main array values; before starting the collection it announces it to other operations, and after the collection it announces its completion. In a second phase, the scan retrieves a linearizable view of the array by combining the values it collected with newer values, forwarded to the designated second array by concurrent update operations (namely, each forwarded value is adopted in place of the value that the scan collected from the corresponding cell in the main array). A scan is linearized at the point it announces completing the collection. It might miss values that were written to the main array by some update operations while it was collecting, thus, such operations are retrospectively linearized immediately after the scan's linearization point. We bring the linearization details of update adapted to our context in Section 3.8.1.

Our size operation acts in the spirit of Jayanti's scan to obtain a view of the metadata array, and data-structure operations that update the metadata array (on behalf of their own operation or to help another operation) act in the spirit of Jayanti's update to inform a concurrent size of a new value it might have missed. However, Jayanti's basic idea supports a single scanner. When multiple size operations execute concurrently, we cannot let each size take an independent snapshot of the metadata array, because the linearization point of a

size operation determines the linearization points of updating operations it missed, and concurrent independent size operations might determine contradicting linearization points for concurrent updates. Thus, we need to make sure that concurrent size operations yield the same consistent snapshot of the metadata array.

To this end, we introduce a CountersSnapshot object (on which we detail in Section 3.6.2). Concurrent size operations coordinate with each other through a CountersSnapshot instance, similarly to concurrent snapshot operations in [PT13] that use a shared object to orchestrate taking a snapshot concurrently. A size operation needs to first obtain a CountersSnapshot instance to operate on. At any given point in time, at most one collecting CountersSnapshot instance (in which the collection has not yet completed) is announced. If a size operation observes such an instance, it operates on it, so that it returns the same size as the size that announced this instance. Otherwise, the size operation produces a new instance, announces it and operates on it.

The CountersSnapshot holds a snapshot array for taking a snapshot of the metadata array. size operations that operate on a certain CountersSnapshot instance collect values into its snapshot array (using a CAS from an initial INVALID value to the value obtained from the metadata array), and operations that concurrently update the metadata array forward their values into the snapshot array. After a collection phase, a size operation needs to compute the size based on the counters in the snapshot array. But the array is not stable—updating operations might be still forwarding values. For all size operations that operate on the same CountersSnapshot instance to agree on the same size, we place a size field in CountersSnapshot, initialized to INVALID. The first size operation to compute a size by traversing the snapshot array and then perform a CAS of the size field from INVALID to its computed size, determines the size value. Concurrent size operations will adopt this value. Any value forwarded to a counter in the snapshot array after the thread that determined the size read this counter is ignored (and its related operation is linearized after the size).

3.6.1 SizeCalculator Details

Each transformed data structure holds a SizeCalculator instance associated with it, responsible for calculating the size by holding the metadata and operating on it. The fields of Size-

```
42 class UpdateInfo:
43     int tid
44     long counter
45 class SizeCalculator:
46     long[][] metadataCounters
47     CountersSnapshot countersSnapshot
48 class CountersSnapshot:
49     long[][] snapshot
50     boolean collecting
51     long size
```

Figure 3.4: Classes fields

Calculator (as well as the other classes we use) are detailed in Figure 3.4, and its pseudocode appears in Figure 3.5.

The SizeCalculator object contains two fields: The first is metadataCounters, holding the size metadata—an array with an insertion counter and a deletion counter per thread, with padding between the cells of each thread and the next one so that the counters of the different threads are placed in separate cache lines to avoid false sharing. The second field is countersSnapshot, that holds the most recent CountersSnapshot instance. In its constructor method (appearing in Line 53), SizeCalculator initializes metadataCounters with zeros, and countersSnapshot with a dummy instance with its collecting flag set to false, to signal that it is not collecting and future size operations should use a new instance.

```
52 class SizeCalculator:
53     SizeCalculator():
54         this.metadataCounters = new long[n][PADDING] // implicitly initialized to zeros
55         this.countersSnapshot = new CountersSnapshot()
56         this.countersSnapshot.collecting.setVolatile(false)
57     compute():
58         activeCountersSnapshot = _obtainCollectingCountersSnapshot()
59         _collect(activeCountersSnapshot)
60         activeCountersSnapshot.collecting.setVolatile(false)
61         return activeCountersSnapshot.computeSize()
62     _obtainCollectingCountersSnapshot():
63         currentCountersSnapshot = this.countersSnapshot.getVolatile()
64         if currentCountersSnapshot.collecting.getVolatile():
65             return currentCountersSnapshot
66         newCountersSnapshot = new CountersSnapshot()
67         witnessedCountersSnapshot = this.countersSnapshot.compareAndExchange(
               ↪ currentCountersSnapshot, newCountersSnapshot):
68         if witnessedCountersSnapshot == currentCountersSnapshot:
69             return newCountersSnapshot
70         return witnessedCountersSnapshot // our exchange failed, adopt the value written by a
               ↪ concurrent thread
71     _collect(targetCountersSnapshot):
72         for each tid:
73             for opKind in (INSERT, DELETE):
74                 targetCountersSnapshot.add(tid, opKind, this.metadataCounters[tid][opKind].
                       ↪ getVolatile())
75     updateMetadata(updateInfo, opKind):
76         tid = updateInfo.tid
77         newCounter = updateInfo.counter
78         if this.metadataCounters[tid][opKind].getVolatile() == newCounter - 1:
79             this.metadataCounters[tid][opKind].compareAndSet(newCounter - 1, newCounter)
80         currentCountersSnapshot = this.countersSnapshot.getVolatile()
81         if currentCountersSnapshot.collecting.getVolatile() and
82                 this.metadataCounters[tid][opKind].getVolatile() == newCounter:
83             currentCountersSnapshot.forward(tid, opKind, newCounter)
84     createUpdateInfo(opKind):
85         return new UpdateInfo(threadID, this.metadataCounters[threadID][opKind].getVolatile() + 1)
```

Figure 3.5: SizeCalculator methods

The compute method is called by the size operation of a transformed data structure. It starts with a collection phase in Lines 58–60. First it needs to announce a new collection if there is no ongoing collection. To this end it calls the private method _obtainCollectingCountersSnapshot. The latter returns the most recent CountersSnapshot if this instance is still collecting (Lines 63–65), so that the current compute would cooperate with ongoing compute calls. Otherwise, _obtainCollectingCountersSnapshot tries to place a new CountersSnapshot instance in countersSnapshot using a CAS, and returns the new countersSnapshot value, whether it is an instance placed by itself or an instance placed by another compute call (Lines 66–70). With an activeCountersSnapshot instance in a collecting mode, compute calls the private method _collect (Line 59), to add all metadataCounters values to activeCountersSnapshot. Then, it unsets activeCountersSnapshot's collecting flag. Now that its collection phase is complete, compute computes the size according to the CountersSnapshot instance maintained in activeCountersSnapshot. This is done using the computeSize method of CountersSnapshot, on which we detail in Section 3.6.2.

updateMetadata(UpdateInfo(tid, c), INSERT) is called on behalf of the c-th successful insert operation by thread tid. We describe how the method handles insertions for convenience; the same applies for deletions by passing opKind=DELETE. The method first updates the relevant counter in the metadata array, i.e., metadataCounters[tid][INSERT], to be c (Lines 78–79), using a CAS to avoid overriding concurrent updates. At this point, the metadata reflects the discussed insertion. Then, according to the described-above scheme, updateMetadata should also forward the counter value c to concurrent size operations that take a snapshot of the metadataCounters array and might have missed this value. For that, it performs the following steps: (1) obtain the current collecting CountersSnpashot instance (Line 80); (2) verify it is still collecting (Line 81); (3) obtain the relevant counter from the metadata array and verify it still holds the value c (Line 82); and then finally, if these checks pass, (4) call the forward method of the CountersSnpashot instance obtained in the first step (Line 83). This series of steps is intended to prevent redundant forwarding. Though it is not yet clear now, it guarantees a constant time complexity for the forward method, as we prove in Section 3.8.2.

The last method of SizeCalculator is createUpdateInfo, which is called by insert and delete operations to obtain an UpdateInfo instance for publication to helpers. createUpdateInfo creates an UpdateInfo instance with tid=threadID and counter=c, where threadID is the ID of the calling thread (threadID values are assumed to start from 0, and could be obtained e.g. from a thread-local variable), and c is the current value of the relevant counter (that indicates how many successful operations of the requested kind have been executed by the calling thread so far) plus 1—as the calling thread is about to attempt its c-th operation of this kind.

3.6.2 CountersSnapshot **Details**

A new CountersSnapshot instance is announced in SizeCalculator.countersSnapshot each time a new collection phase starts (which happens every time a size operation starts and observes that the last announced CountersSnapshot instance is already not collecting). This instance coordinates the current size calculation among all concurrent size calls that use it to compute the size. Its methods appear in Figure 3.6 and its fields appear in Figure 3.4.

The CountersSnapshot object holds a snapshot array called snapshot for taking a snapshot of the metadata array, from which the size will be computed. It also holds a collecting field that indicates whether the collection of values into snapshot is still ongoing, and a size field that will eventually hold the computed size. In its constructor method (appearing in Line 87), CountersSnapshot initializes all its fields. The cells of snapshot are set to INVALID (which may have the value Long.MAX_VALUE for instance), the collecting flag is set to true and size is set to INVALID.

The add method is called by size operations to collect values into the snapshot array. It performs a CAS on the requested cell to the requested value only if the current value is INVALID. Otherwise, another operation has already collected a value to this cell and there is no need to perform another CAS. Indeed, the value that this size operation fails to add might be missed during the size calculation if it is not forwarded on time by the updating operation

```
86  class CountersSnapshot:
87      CountersSnapshot():
88          this.snapshot = new long[n][2]
89          setVolatile all cells of this.snapshot to INVALID
90          this.collecting.setVolatile(true)
91          this.size.setVolatile(INVALID)
92      add(tid, opKind, counter):
93          if this.snapshot[tid][opKind].getVolatile() == INVALID:
94              this.snapshot[tid][opKind].compareAndSet(INVALID, counter)
95      forward(tid, opKind, counter):
96          snapshotCounter = this.snapshot[tid][opKind].getVolatile()
97          while (snapshotCounter == INVALID or
                    counter ⟩snapshotCounter): // will execute at most two iterations
98              witnessedSnapshotCounter = this.snapshot[tid][opKind].compareAndExchange(
                    ↪ snapshotCounter, counter):
99              if witnessedSnapshotCounter == snapshotCounter:
100                 break
101             snapshotCounter = witnessedSnapshotCounter
102     computeSize():
103         computedSize = 0
104         for each tid:
105             computedSize += this.snapshot[tid][INSERT].getVolatile() -
                        this.snapshot[tid][DELETE].getVolatile()
106         witnessedSize = this.size.compareAndExchange(INVALID, computedSize)
107         if witnessedSize == INVALID:
108             return computedSize
109         return witnessedSize // our exchange failed, adopt the size written by a concurrent thread
```

Figure 3.6: CountersSnapshot methods

associated with it, but this does not foil linearizability, as the updating operation associated with this value is retrospectively linearized after the size.

forward(tid,INSERT,c) is called by updateMetadata that was called on behalf of the c-th successful insert operation by thread tid. It is called after the insertion counter of thread tid in the metadata array is set to c, to ensure that the insertion counter of that thread in the snapshot array contains a value $\geq c$. We prove in Section 3.8.2 that the forward method shall execute at most two CAS attempts before reaching this goal. forward operates similarly for deletions when called with an opKind=DELETE argument.

The computeSize method is called by the compute method of SizeCalculator (which is called by the data structure's size method), after obtaining a CountersSnapshot instance and completing the collection to this instance, so that its snapshot array is filled with meaningful values (rather than INVALID values). The size is computed as the difference between the sum of insertion counters and the sum of deletion counters in the snapshot array (Lines 103–105). But computeSize may be called by multiple concurrent size operations that operate on the same CountersSnapshot instance, and each of them might compute a different size because values may be concurrently forwarded to the array. Only the first computeSize call to fix the size it computed in the size field (in Line 106), determines the size value that they will all adopt. The rest of them will fail to CAS and will adopt its value.

3.6.3 Memory Model

The pseudocode brought in this section aligns with our Java implementation [SP22c] (evaluated in Section 3.9) and accesses variables using volatile memory semantics to ensure the visibility required for correctness in accordance with the Java memory model. Read, write and CAS operations on non-final fields of shared objects are performed with volatile semantics (in our Java implementation this is achieved using volatile variables, VarHandles and AtomicReferenceFieldUpdaters). These volatile-semantics accesses are considered synchronization actions, over which the Java memory model guarantees a synchronization order (a total order which is consistent with the program order of each thread, and where a read from a volatile variable returns the last value written to it by the synchronization order). A similar implementation could be designed in C++ according to its memory model guarantees, utilizing the std::atomic library to order accesses to shared memory.

3.7 Optimizations

The following optimizations may be applied in our methodology, and we apply them in our implementation [SP22c] measured in Section 3.9.

3.7.1 Eliminate Metadata Update on Behalf of Completed Insertions

When an insertion is complete, there is no need that future operations on the inserted node update the metadata on behalf of that insertion. To this end, after a thread calls updateMeta-

data to update the metadata on behalf of some insert operation that inserted a node N, it may set N.insertInfo to NULL, to signal that the metadata already reflects the insertion and there is no need to call updateMetadata. Before calling updateMetadata, threads will perform a NULL check to the node's insertInfo to rule if the call is necessary.

We do not propose a similar modification for deletions since deleted nodes are unlinked from the data structure when the deletion completes and cause no more update activity, unlike inserted nodes which, without the optimization, cause a redundant updateMetadata call on each operation on the node.

3.7.2 Size Backoff

Each size operation operates on a CountersSnapshot instance it obtains as follows. It collects values into its snapshot array using CAS operations, uses the collected counters to compute the size, and finally sets its size field to the computed size using a CAS, unless another size operation has done that beforehand.

Exponential backoff may be used to reduce contention among concurrent size operations caused by their CAS operations on the snapshot and size fields. Each time a size operation obtains an existing CountersSnapshot instance that was announced by another size operation, it may wait a while to let another size operation complete the size calculation. After waiting, if the calculation is not yet complete (which may be detected by an INVALID value in the size field), it shall collect, compute the size and try to set it on its own.

3.7.3 Check for an Already-Set Size

There are occasions where we may avoid contention and redundant work by obtaining the size field of CountersSnapshot and returning it in case it does not equal INVALID. This may be done when SizeCalculator's _obtainCollectingCountersSnapshot method observes a concurrent size operation in Lines 65 and 70; at the beginning of CountersSnapshot's computeSize method; and right before computeSize performs a CAS attempt.

3.8 Methodology Properties

3.8.1 Linearizability

A linearizable data structure transformed according to our methodology to support a size operation, remains linearizable. Recall that an operation has its original linearization point, when its linearization is defined in the original set data structure, but we linearize operations in the transformed data structure only when the metadata is updated. Next, we detail the linearization points of a transformed set's operations, and use them to prove linearizability.

Linearization Points

A size operation is linearized at the announcement of the collection completion. For a successful insert or delete operation, the associated metadata counter is updated to reflect the operation (by either the operation initiator or helpers), and if this update happens when no size is collecting, then the operation is simply linearized at the update. However, if the update is performed while some size is collecting, then the operation is linearized according to that size to comply with its linearization point: if the size takes the operation into account then the operation is simply linearized at the metadata counter update; otherwise, it is retrospectively linearized immediately after the linearization point of that size. Finally, a contains operation and a failing insert or delete operation (namely, one that fails to insert a new key or delete an existing key respectively, and returns a failure), are linearized like in the original data structure, unless the operation they "depend on", namely, the last successful update operation on the same key whose original linearization point precedes their original linearization point (a concurrent successful insert of the same key in case of contains returning true and a failing insert; and a concurrent successful delete in case of contains returning false and a failing delete) is not yet linearized at their original linearization point, in which case they are linearized immediately after this operation is linearized.

In more detail, a size operation is linearized when the collecting field of the CountersSnapshot instance it operates on is set to false for the first time (in Line 60). Regarding a successful insert operation, the associated metadata counter is updated as follows: a CAS of sizeCalculator.metadataCounters[tid][INSERT] to c is performed on behalf of the c-th successful insert operation of thread tid (in Line 79), where sizeCalculator is the SizeCalculator instance held by the transformed data structure. For a successful delete operation, the only difference is that DELETE is used as the array index. As for the linearization point of such an insert or delete operation—if a CountersSnapshot instance with a collecting field set to true is not announced in sizeCalculator.countersSnapshot when the metadata counter update is performed, then the operation is linearized at the metadata counter update (namely, at the CAS in Line 79). Else, the operation is linearized according to this CountersSnapshot instance: if the size operation that sets its size field read a value $\geq c$ from the relevant counter (in Line 105), then the operation is linearized at the metadata counter update (as in the previous case); otherwise, it is linearized immediately after the linearization point of that size operation.

In the above specification, several operations might be linearized at the same moment—either operations defined to be linearized immediately after each other, or operations linearized at the exact same moment (e.g., several size operations operating on the same CountersSnapshot instance). We order operations that are linearized at the same moment one after the other as follows: size operations (if any) are placed first; the order among them is arbitrary. Successful update operations (if any) are placed after the size operations according to their metadata-counter update order. Each contains or failing insert or delete call that is not linearized at its original linearization point (if any) is placed right after the successful

update operation it depends on; the order among such operations which are placed after the same successful update is arbitrary.

Linearizability Proof

We prove that our transformation is linearizable using the equivalent definition of linearizability that is based on linearization points (see [SHP21b, Section 7] and the atomicity definition in [Lyn96]). We need to show that (1) each linearization point occurs within the operation's execution time, and that (2) ordering an execution's operations (with their results) according to their linearization points forms a legal sequential history.

We prove Property (1) in Claim 3.8.1 and Property (2) in Claim 3.8.4.

Claim 3.8.1. *The linearization point of each operation occurs within its execution time.*

Proof We begin with the linearization point of a size operation. size calls sizeCalculator.compute, which starts with calling _obtainCollectingCountersSnapshot to obtain a collecting CountersSnapshot instance. _obtainCollectingCountersSnapshot returns an instance after its collecting field has had the value true at some point during this _obtainCollectingCountersSnapshot call: If this call observes that the current announced instance is collecting (in Line 64), it returns this instance. Otherwise, this instance cannot be used by the current size because its linearization point has passed and has possibly occurred before the current size started. Thus, it creates an instance with collecting set to true, and if it succeeds to announce it using a CAS (in Line 67), it returns this instance. Else, the failure of its CAS indicates that another thread has in the meanwhile announced a new instance, with collecting set to true, and the discussed _obtainCollectingCountersSnapshot call returns such an instance. We showed that in any of the above cases, the collecting field of the obtained CountersSnapshot instance was still true at some point during the _obtainCollectingCountersSnapshot call, hence the size's linearization point does not occur before the compute call starts. It does occur before it ends, as the collecting field is set to false either when this call executes Line 60, or before if another compute call has executed this line earlier.

Next, we prove that successful update operations are linearized within their execution time. A successful insert or delete operation calls updateMetadata with its UpdateInfo instance before returning. As we prove in Lemma 3.8.2 below, by the time updateMetadata returns, the operation is guaranteed to be linearized. Additionally, it is not linearized before the operation's execution starts, since it is linearized either at its metadata counter update or at a later point in time, and the update on behalf of a certain operation can only happen after it started and published its UpdateInfo instance.

Lastly, we show that a contains, a failing insert and a failing delete operations are linearized within their execution time. If an operation *op* of this kind is linearized at its original linearization point, we are done[1]. Otherwise, *op* is linearized immediately after the linearization point of an operation op_2 it depends on. This happens only in case *op* observes op_2 and

[1]For every linearizable data structure, there exists a selection of linearization points such that each of them is placed during the execution time of the corresponding operation (see the equivalent definition of linearizability

calls updateMetadata on behalf of op_2. By Lemma 3.8.2, op_2 is linearized by the time this updateMetadata call returns. Hence, *op* is linearized by that time as well.

In the proof of Claim 3.8.1 we use the following lemmas:

Lemma 3.8.2. *When a call to* updateMetadata *returns, the operation whose* updateInfo *was passed to the call is guaranteed to be linearized.*

Proof Consider a call to updateMetadata on behalf of *op*, being the c-th successful insert operation by a thread T (a similar proof applies for delete). Denote this call by *updateMetadataForOp*. We need to show that *op* has been linearized by the time *updateMetadataForOp* returns. By Lemma 3.8.3, after executing Lines 78–79, the relevant metadata counter's value is $\geq c$. If *op* is linearized when its related metadata counter is set to c, we are done. Otherwise, the following hold: (1) the counter is set to c when a CountersSnapshot instance with a collecting field set to true is announced in the countersSnapshot field of sizeCalculator (denote this instance by *snapshotAtUpdate*); (2) the size operation that sets *snapshotAtUpdate*'s size field reads, during its size computation, a value $< c$ from the corresponding snapshot counter; and (3) *op* is linearized immediately after the linearization point of that size operation, namely, immediately after the collecting field of *snapshotAtUpdate* is set to false for the first time. So we need to prove that this collecting field is set to false before the *updateMetadataForOp* call returns. Next, we prove this holds in the various possible scenarios.

If *updateMetadataForOp* obtains a newer CountersSnapshot instance than *snapshotAtUpdate* in Line 80, then we are done, as CountersSnapshot instances announced in SizeCalculator are replaced only after their collecting field is set to false.

Else, if *updateMetadataForOp* observes in Line 81 that *snapshotAtUpdate*'s collecting field value is false, we are done.

Else, if the checks in Lines 81 and 82 pass, *updateMetadataForOp* forwards the value c to the snapshot counter in Line 83. When its forwarding completes, the snapshot counter contains a value $\geq c$. Since we analyze here a case in which the size operation, which sets *snapshotAtUpdate*'s size field, reads a value $< c$ from the snapshot counter, this size must have read the snapshot counter before the forwarding completes, and this read during the size computation occurs only after the *snapshotAtUpdate*'s collecting field is set to false.

The remaining scenario is that *updateMetadataForOp* observes in Line 82 that the metadata counter's value is $\geq c + 1$. We will show that *snapshotAtUpdate*'s collecting field value has been earlier set to false. For the metadata counter to reach the value $c + 1$, the thread T must have already started its $(c + 1)$-st successful insertion. Prior to that, T has completed *op* (which is its c-th successful insertion), during which it has called updateMetadata, in a call we denote *updateMetadataByT*. We will next show that by the time *updateMetadataByT* returns, the value of *snapshotAtUpdate*'s collecting field is already false. After *updateMetadataByT* executes Lines 78–79, the metadata counter's value is $\geq c$ by Lemma 3.8.3. Denote by

based on linearization points in Section 7 in [SHP21b]). Each time we refer to the linearization points of the original data structure, we refer to points that satisfy this requirement.

currSnap the value that *updateMetadataByT* obtains in currentCountersSnapshot in Line 80. *currSnap* must be *snapshotAtUpdate*, because otherwise, if it were an earlier CountersSnapshot instance, then when *snapshotAtUpdate* is later announced in the countersSnapshot field of sizeCalculator, the metadata counter's value would already be $\geq c$ as mentioned above, but this contradicts Attribute (2) above which implies that a value $< c$ is collected in *snapshotAtUpdate*. Thus, *currSnap* is *snapshotAtUpdate*. When *updateMetadataByT* checks the value of *snapshotAtUpdate*'s collecting field in Line 81, if it is false then we are done. Otherwise, *updateMetadataByT* calls the forward method to ensure that the value c is forwarded to the snapshot counter. Like in the previous scenario, we analyze here a case in which the size operation, which sets *snapshotAtUpdate*'s size field, reads a value $< c$ from the snapshot counter, hence, this size must have read the snapshot counter before the forwarding completes, and this read during the size computation occurs only after the *snapshotAtUpdate*'s collecting field is set to false. This concludes the proof.

Lemma 3.8.3. *Consider a call to* updateMetadata *on behalf of op, being the c-th successful* insert *or* delete *operation by a thread T. After this call executes Lines 78–79, the relevant metadata counter's value is $\geq c$.*

Proof We prove by induction. Assume the lemma holds for $c-1$. The metadata counters are modified only in Line 79 by increments using CAS. Hence, it is enough to prove that when the updateMetadata call starts, the relevant metadata counter's value is $\geq c-1$. updateMetadata is called with an UpdateInfo instance associated with op after this instance has been published in the relevant node. This publication is done by the thread T when it executes op, and as each thread executes its operations sequentially, T has completed its $(c-1)$-st successful operation of the same kind (insertion or deletion) by this time. During that operation, T called updateMetadata on its behalf, so by the induction hypothesis, the relevant metadata counter's value is $\geq c-1$ when T completes that previous operation. Since the counters are monotonically increasing, we are done. ■

To complete the linearizability proof, it remains to prove Claim 3.8.4. In what follows, we denote the set's i-th successful $\mathrm{insert}(k)$ operation (by *i-th* we refer to the linearization order, namely, to the i-th successful $\mathrm{insert}(k)$ to be linearized) by $\mathrm{insert}_i(k)$, its linearization time by $t_{insert_i(k)}$, and the time of its original linearization by $orig\_t_{insert_i(k)}$. We further denote the analogous delete operation and its related times by $\mathrm{delete}_i(k)$, $t_{delete_i(k)}$ and $orig\_t_{delete_i(k)}$.

Claim 3.8.4. *Consider a sequential history formed by ordering an execution's operations (with their results) according to their linearization points defined in Section 3.8.1. Then operation results in this history comply with the sequential specification of a set.*

Proof As for the results of successful update operations, their correctness follows directly from Corollary 3.1: The last successful update operation on k to be linearized before the linearization point of a successful $\mathrm{insert}(k)$ operation is a deletion, thus, the key k is logically not

in the set at the moment of the insertion's linearization and the insertion correctly succeeds. Similarly, the last successful update operation on k to be linearized before the linearization point of a successful $\mathrm{delete}(k)$ operation is an insertion, thus, the key k is logically in the set at the moment of the deletion's linearization and the deletion correctly succeeds.

Now, let us examine the results of $\mathrm{contains}$ operations and failing update operations. Let *op* be such an operation on a key k, and let the operation it depends on (namely, the last successful update operation on k whose original linearization point precedes *op*'s original linearization point) be $\mathrm{insert}_i(k)$ for some $i \geq 1$ (the proof for a delete operation is similar). As $\mathrm{insert}_i(k)$ is an insertion, *op* must be a $\mathrm{contains}$ operation returning true or a failing insert operation. To show that *op*'s result—which reflects that the last operation on the set was an insertion—is legal, we will prove that the linearization point of *op* occurs when the last linearized successful update operation on k is the insertion $\mathrm{insert}_i(k)$. Let $orig\_t_{op}$ be the original linearization moment of *op*. There are two possibilities with regards to *op*'s linearization point: either *op* is linearized immediately after $t_{insert_i(k)}$, and we are done, or it is linearized at $orig\_t_{op}$. In the latter case, according to the linearization point definition, $\mathrm{insert}_i(k)$ must be linearized by *op*'s original linearization moment, namely, $t_{insert_i(k)} < orig\_t_{op}$. If no successful $\mathrm{delete}(k)$ operation is linearized after $t_{insert_i(k)}$, then we are done. Else, $orig\_t_{op} < orig\_t_{delete_i(k)}$, as $\mathrm{insert}_i(k)$ is the last successful update operation on k whose original linearization point precedes $orig\_t_{op}$. Since $orig\_t_{delete_i(k)} < t_{delete_i(k)}$ (by Lemma 3.8.6), then $orig\_t_{op} < t_{delete_i(k)}$, and we are done.

Finally, we analyze the linearization of a size operation. Denote such an operation by *op*, the $\mathrm{CountersSnapshot}$ instance it obtains and operates on by *countersSnapshot*, and the size call that sets the *countersSnapshot*.size field by *determiningSize*. *op* returns the difference between the sum of insertion counters and the sum of deletion counters that were observed in the *countersSnapshot*.$\mathrm{snapshot}$ array by *determiningSize*. Let j be the value that *determiningSize* obtained from the insertion counter of some thread T in *countersSnapshot*.$\mathrm{snapshot}$. We will prove that T's j-th successful insert is linearized before *op*'s linearization point, and T's $(j+1)$-st successful insert (if such an operation occurs) is linearized after it. We refer to insertions for convenience, but the exact same proof applies to the deletion counters as well.

We start with T's j-th successful insert. Since *determiningSize* obtained the value j from the relevant snapshot counter, then by Lemma 3.8.7, the metadata counter update on behalf of T's j-th successful insert happens before *op*'s linearization point. If T's j-th successful insert is linearized in its metadata counter update, we are done. Else, it is linearized immediately after the linearization point of a size operation, whose collecting $\mathrm{CountersSnapshot}$ instance is announced when the metadata counter is updated, and which reads a value $< j$ from the relevant snapshot counter. This size operation cannot be *op* (which reads the value j), but rather a preceding size operation whose $\mathrm{CountersSnapshot}$ instance is announced prior to $\mathrm{countersSnapshot}$ (because it is already announced when the metadata counter is updated on behalf of T's j-th successful insert, which by Lemma 3.8.7 happens before *countersSnapshot*'s $\mathrm{collecting}$ field is set to false), so its linearization point precedes *op*'s linearization point.

We proceed to T's $(j+1)$-st successful insert (in case such an operation occurs). If in its

metadata counter update, *countersSnapshot* is announced in the $\mathrm{SizeCalculator}$ instance held by the set and its $\mathrm{collecting}$ field's value is true, then this insertion is linearized immediately after *op*'s linearization point, because *determiningSize* obtained the value j (which is smaller than $j+1$) from the corresponding *countersSnapshot*.$\mathrm{snapshot}$'s counter. Else, this metadata counter update must have occurred after *op*'s linearization point, since the alternative is that *countersSnapshot* is announced after that metadata counter update, in which case a value $\geq j+1$ must be collected in *countersSnapshot*.$\mathrm{snapshot}$. The insertion is linearized either at its metadata counter update or later, thus, linearized after *op*'s linearization point in this case as well. ■

The proof of Claim 3.8.4 uses the following:

Observation 3.8.5. The original linearization points of successful insertions and deletions of each key k are alternating.

This follows from the linearizability of the original data structure and the sequential specification of a set.

Lemma 3.8.6. *For each key k and each $i \geq 1$:*

$$orig\_t_{insert_i(k)} < t_{insert_i(k)} < orig\_t_{delete_i(k)} < t_{delete_i(k)} < orig\_t_{insert_{i+1}(k)}$$

Proof The linearization point of each successful insert or delete operation happens after its original linearization point because the linearization point occurs at the metadata counter update or later, and this update is performed in our transformation after the original linearization point.

In addition, before $\mathrm{delete}_i(k)$ carries out its own original linearization point, i.e., marking the node it is deleting, it calls $\mathrm{updateMetadata}$ on behalf of the insert operation that inserted that node. By Observation 3.8.5, the last original linearization point of a successful update operation on k before the one of $\mathrm{delete}_i(k)$ is that of $\mathrm{insert}_i(k)$. Thus, the node that $\mathrm{delete}_i(k)$ deleted was inserted by $\mathrm{insert}_i(k)$, and $\mathrm{delete}_i(k)$ calls $\mathrm{updateMetadata}$ with the $\mathrm{insertInfo}$ associated with $\mathrm{insert}_i(k)$. By Lemma 3.8.2, $\mathrm{insert}_i(k)$ will have been linearized by the time this $\mathrm{updateMetadata}$ call returns. Hence, $t_{insert_i(k)} < orig\_t_{delete_i(k)}$.

It remains to prove that $t_{delete_i(k)} < orig\_t_{insert_{i+1}(k)}$. By Observation 3.8.5, $\mathrm{insert}_{i+1}(k)$'s original linearization point occurs after $\mathrm{delete}_i(k)$'s original linearization point. If the node deleted by $\mathrm{delete}_i(k)$ has been already unlinked prior to $orig\_t_{insert_{i+1}(k)}$, then prior to the unlinking, $\mathrm{updateMetadata}$ has been called on behalf of $\mathrm{delete}_i(k)$. Else, we note that in all set implementations we are aware of, if at the original linearization moment of a successful $\mathrm{insert}(k)$ there exists a node with the key k reachable from the data structure's roots, then the insert operation must have observed this node earlier, during its search for k. Thus, $\mathrm{insert}_{i+1}(k)$ observes the node deleted by $\mathrm{delete}_i(k)$, and calls $\mathrm{updateMetadata}$ on its behalf before carrying out its own original linearization point. In both cases, by Lemma 3.8.2, $\mathrm{delete}_i(k)$ is linearized by the time the $\mathrm{updateMetadata}$ call returns, thus, linearized before $orig\_t_{insert_{i+1}(k)}$. ■

Corollary 3.1. *The linearization points of successful insertions and deletions of each key k are alternating.*

Lemma 3.8.7. *Let countersSnapshot be a* CountersSnapshot *instance. Any non-*INVALID *value written to a counter in the countersSnapshot.*snapshot *array must have been written to the corresponding counter in the* metadataCounters *array (of the* SizeCalculator *instance held by the set)* before *the countersSnapshot.*collecting *field is set to* false.

Intuitively, this implies that a size operation cannot witness future update operations (namely, ones that are linearized after it).

Proof Consider some *countersSnapshot.*snapshot's cell C of either an insertion or a deletion counter of some thread T. We will analyze all possible writes of non-INVALID values to C, and show that they write values that have been written to T's corresponding metadata counter before *countersSnapshot.*collecting is set to false. Non-INVALID values are written to *countersSnapshot.*snapshot in the CountersSnapshot's add and forward methods, in Lines 94 and 98 respectively. Starting with add, only the first execution of the CAS in Line 94 on C succeeds, and it occurs within a call to SizeCalculator's _collect method, before the first time *countersSnapshot.*collecting is set to false in Line 60. As for forward, it is called by SizeCalculator's updateMetadata method for forwarding some value val to *countersSnapshot* after (1) val is written to the relevant T's metadata counter—as guaranteed by Lemma 3.8.3, and then (2) *countersSnapshot.*collecting is verified to bear the value true in Line 81. Hence, val has been written to the relevant metadata counter before *countersSnapshot.*collecting is set to false. ■

3.8.2 Wait-Freedom and Asymptotic Time Complexity

The size operation in our methodology is wait-free as it completes within a constant number of steps, regardless of other threads' progress. Its time complexity is linear in the number of threads due to its two passes on arrays with per-thread counters: during the collection process (in the _collect method) and the size computation (in the computeSize method).

Our transformation preserves the time complexity and progress guarantees of the data-structure operations, as each call to the udpateMetadata method adds a constant number of steps. This follows from the following claim.

Claim 3.8.8. *Each call to the* forward *method of* CountersSnapshot *(by* updateMetadata*) executes at most two iterations of its while loop.*

Before forwarding, updateMetadata performs several checks in a certain order. Without this careful procedure, delayed threads that run updateMetadata to help old operations could forward stale values to the snapshot array, causing an updateMetadata on behalf of a recent operation to repeatedly fail forwarding. Next, we show how this procedure prevents forwarding stale values.

Proof Consider a call to updateMetadata that calls forward and operates on behalf of *op*, being the c-th successful insert operation by a thread T (a similar proof applies for delete). Denote by *currSnap* the value this call obtains in currentCountersSnapshot (in Line 80) at time denoted t_{obt}. As the snapshot counters are monotonically increasing, it is enough to prove that from t_{obt} and on, only values $\geq c-1$ may be written to the snapshot counter of *currSnap* that is associated with *op*.

Note that after obtaining *currSnap* at time t_{obt} and before calling forward, updateMetadata observes that *currSnap* is in a collecting mode (in Line 81), thus, it has been in this mode at t_{obt}. Now, let t_{c-1} be the time in which the metadata counter associated with *op* is set to $c-1$. If *currSnap* has been announced in sizeCalculator.countersSnapshot before time t_{c-1}, then it keeps being announced and in collecting mode at least until time t_{obt}. Thus, the value $c-1$ is forwarded to the snapshot counter associated with *op* before time t_{obt}, as otherwise the thread T would not have proceeded from its $(c-1)$-st successful insert to its c-th successful insert, and we are done since the snapshot counter is monotonically increasing.

Otherwise, *currSnap* is announced in sizeCalculator.countersSnapshot after time t_{c-1}. Two methods write to the snapshot array: add and forward. add is called (in Line 74) with a counter value that is obtained from the metadata array after *currSnap* is announced, hence, after time t_{c-1}, so it writes a value $\geq c-1$. As for forward, a thread that calls it (in Line 83) to forward a value to the snapshot counter associated with *op*, performs the following steps in this order: (1) obtain *currSnap* in Line 80—which must happen after *currSnap* is announced and hence after time t_{c-1}; (2) obtain the value of the metadata counter associated with *op* in Line 82; and then (3) forward this value to *currSnap*'s snapshot array. Since the value is obtained after time t_{c-1}, it must be $\geq c-1$. ■

3.9 Evaluation

In this section, we present the evaluation of our methodology on several data structures. The code for the data structures and the measurements is available at [SP22c]. We first measure the overhead that the addition of the size mechanism introduces to operations of transformed data structures (in Section 3.9.1 we break down the overhead by operation type). Then, we demonstrate the benefits of computing a linearizable size in our methodology. We show that it yields a performance better in orders of magnitude than the alternative of taking a linearizable snapshot of the data structure and counting its elements. We further demonstrate the scalability of our methodology and its performance insensitivity to the data-structure size.

Evaluated data structures We start with three baseline data structures that do not support a linearizable size: a skip list, a hash table and a binary search tree, denoted SkipList, HashTable and binary search tree (BST) respectively. The implementation of SkipList is taken from the ConcurrentSkipListMap class of the java.util.concurrent package of Java SE 18. We eliminated methods irrelevant to our measurements and kept the main insert, delete and contains functionality. As for a hash table, we implemented HashTable as a table

of linked lists whose implementation is based on the linked list in the base level of SkipList. We use a table of a static size (chosen in a way similar to Java's ConcurrentHashMap to be a power of 2 between $1\times$ and $2\times$ the number of elements; we detail below how we keep the number of elements stable during the measurements). We do not use Java's lock-based ConcurrentHashMap as our hash-table baseline because it deletes items by unlinking without marking (as it does not use a delicate synchronization mechanism but rather coarse-grained locking), thus, our transformation is not applicable to it as is. For BST we use Trevor Brown's implementation [Bro18] of the lock-free binary search tree of [EFRvB10] that places elements in leaf nodes.

We applied our methodology to each of the baseline algorithms, to produce the transformed data structures SizeSkipList, SizeHashTable and size-supporting binary search tree (SizeBST) that support a linearizable size. In the case of the tree, BST linearizes the delete operation at the unlinking and not in the prior marking of the deleted node's parent. Hence, we formed a variant of BST that linearizes delete at the marking step, and then applied our methodology to this variant.

We compare performance of the size operation in the data structures produced using our methodology with a snapshot-based size operation in two data structures supporting snapshots. The first one is SnapshotSkipList—[PT13]'s implementation of a skip list with a snapshot mechanism, which we obtained from the paper's authors. Like our skip list implementations (SkipList and SizeSkipList), it builds on Java's ConcurrentSkipListMap. It uses code from an older version of Java, but the performance degradation incurred by the older version is negligible and irrelevant to our measurements, due to the immense performance difference between obtaining the size using their snapshot and using our methodology. The size operation is implemented in SnapshotSkipList by taking a snapshot, which produces a snapshot copy of the base level of the skip list, and then iterating it and counting its elements.

The second competitor we compare to is VcasBST-64 [WBB+21]—a binary search tree with a snapshot mechanism taken from the paper's published implementation [Wei21]. It is based on the same implementation by Brown that BST and SizeBST are based on, but uses a modified version of it which batches multiple keys in leaves and stores up to 64 key-value pairs in each tree leaf. To compute the size, we did not use their implementation as a black box, as their interface supplies a snapshot copy of the tree's elements, but such copying is redundant for retrieving the size. Instead, to compute the size we call their snapshot operation that advances the timestamp, and then traverse the tree and sum the number of elements in leaves with a timestamp no bigger than the snapshot timestamp. By this, we save copying all tree elements, and even save iterating the elements one by one—as we simply read the leaf's number of elements (each batched leaf node keeps the number of contained elements). Even though we use this improved size implementation for VcasBST-64, and even though VcasBST-64 uses batched leaves which enables it to perform faster than without them, we will show that our size computation method still outperforms it.

Platform We conducted our experiments on a machine running Linux (Ubuntu 20.04) equipped with 4 AMD Opteron(TM) 6376 2.3 GHz processors. Each processor has 16 cores, resulting in 64 threads overall. The machine used 64 GB RAM, an L1 data cache of 16 KB per core, an L2 cache of 2 MB for every two cores, and an L3 cache of 6 MB for every 8 cores.

The implementations were written in Java. We used OpenJDK 17.0.2 with the flags -server, -Xms15G and -Xmx15G. The latter two flags reduce interference from Java's garbage collection. We used the G1 garbage collector (using ParallelGC yields similar results).

Methodology Before each experiment, we fill the data structure with 1M items, except for the experiments that check dependence on the data-structure size, in which we fill the data structure with a varying number of items—1M, 10M or 100M. We chose these sizes in order to measure the performance of the data-structure when it does not fit into the L3 cache.

We run two workloads: an update-heavy workload, with 30% insert operations, 20% delete operations and 50% contains operations, and a read-heavy workload, with 3% insert operations, 2% delete operations and 95% contains operations. These workloads match the read rates suggested by *Yahoo! Cloud Serving Benchmark* (YCSB) [CST+10]—update-heavy workloads with 50% reads and read-heavy workloads with 95% reads (YCSB also suggests a 100%-read workload, but this is less relevant to our case, since it is less likely to have size calls on a data structure that never changes). The left part of Figures 3.7–3.13 shows results for the read-heavy workload, and the right part shows results for the update-heavy workload.

Similarly to [WBB+21], keys for operations during the experiment and for the initial filling are drawn uniformly at random from a range $[1, r]$, where r is chosen to maintain the initial size of the data structure. For example, for $n = 1$M initial keys and a workload with 30% inserts and 20% deletes, we use $r = n \cdot (30 + 20)/30 \approx 1.67M$.

In all experiments except for the experiments that check how overhead is split by operation type, we repeatedly choose (by the update-heavy or read-heavy proportion) the type of the next operation. However, in the overhead-split measurements (that appear in Section 3.9.1), we repeatedly choose a uniform type for the next 100 operations, because in these measurements we need to obtain the time it took to execute operations of each type, and obtaining the time it took to execute too few consecutive operations of the same type would impair the time measurement accuracy.

In each experiment, we run w workload threads, performing insert, delete and contains calls according to the update-heavy or read-heavy workloads, and s size threads, repeatedly calling size, except for executions of the baseline algorithms (HashTable, SkipList and BST—evaluated in the overhead and overhead breakdown measurements), for which we run w workload threads only. w and s vary across experiments, and we took $w + s$ to be a power of 2 in most experiments. In each experiment, the threads perform operations concurrently for 5 seconds. Each data point in the graphs represents the average result of 10 runs, after executing 5 preliminary runs to warm up the Java virtual machine. The coefficient of variation was up to 11% in the experiments we present next, and up to 21% in the experiments presented in Section 3.9.1.

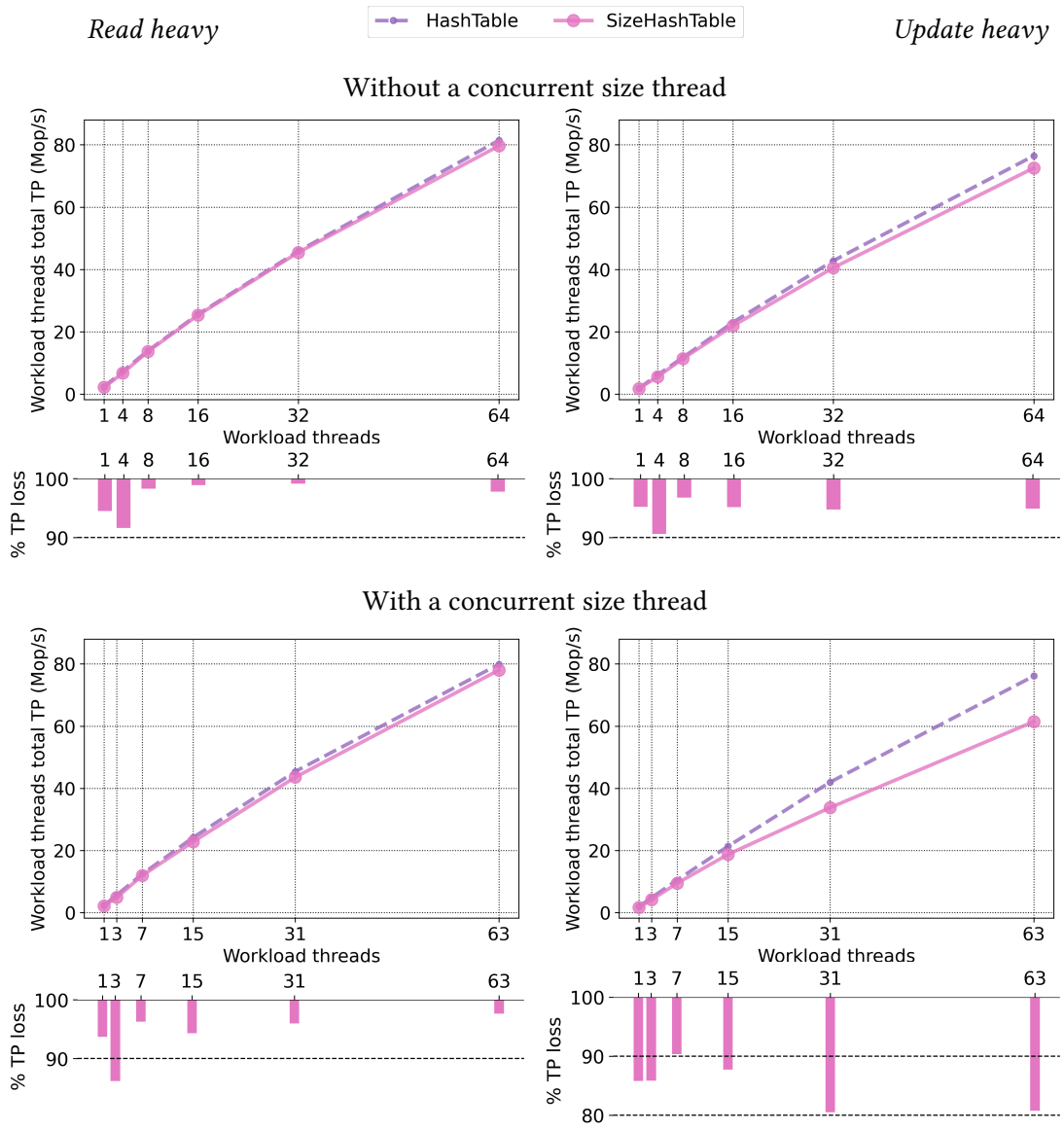


Figure 3.7: Overhead on hash table operations

Overhead We measure the overhead of our methodology on the original data-structure operations by measuring the performance of workload threads—executing insert, delete and contains operations. We compare the total throughput of w workload threads, where w varies from 1 to 64, for the transformed data structures versus the baseline data structures. The results appear in the top part of Figures 3.7–3.9: the results for SizeHashTable versus HashTable appear in Figure 3.7, for SizeBST versus BST in Figure 3.8, and for SizeSkipList versus SkipList in Figure 3.9. They show the overhead when no concurrent size operations are executed. To measure the overhead in the presence of size calls as well, we similarly run w workload threads, where w varies from 1 to 63, while also running—for the transformed algorithms only—a concurrent size thread (that executes size calls), and measure the total throughput of the workload threads. The results of these experiments appear in the bottom part of Figures 3.7–3.9.

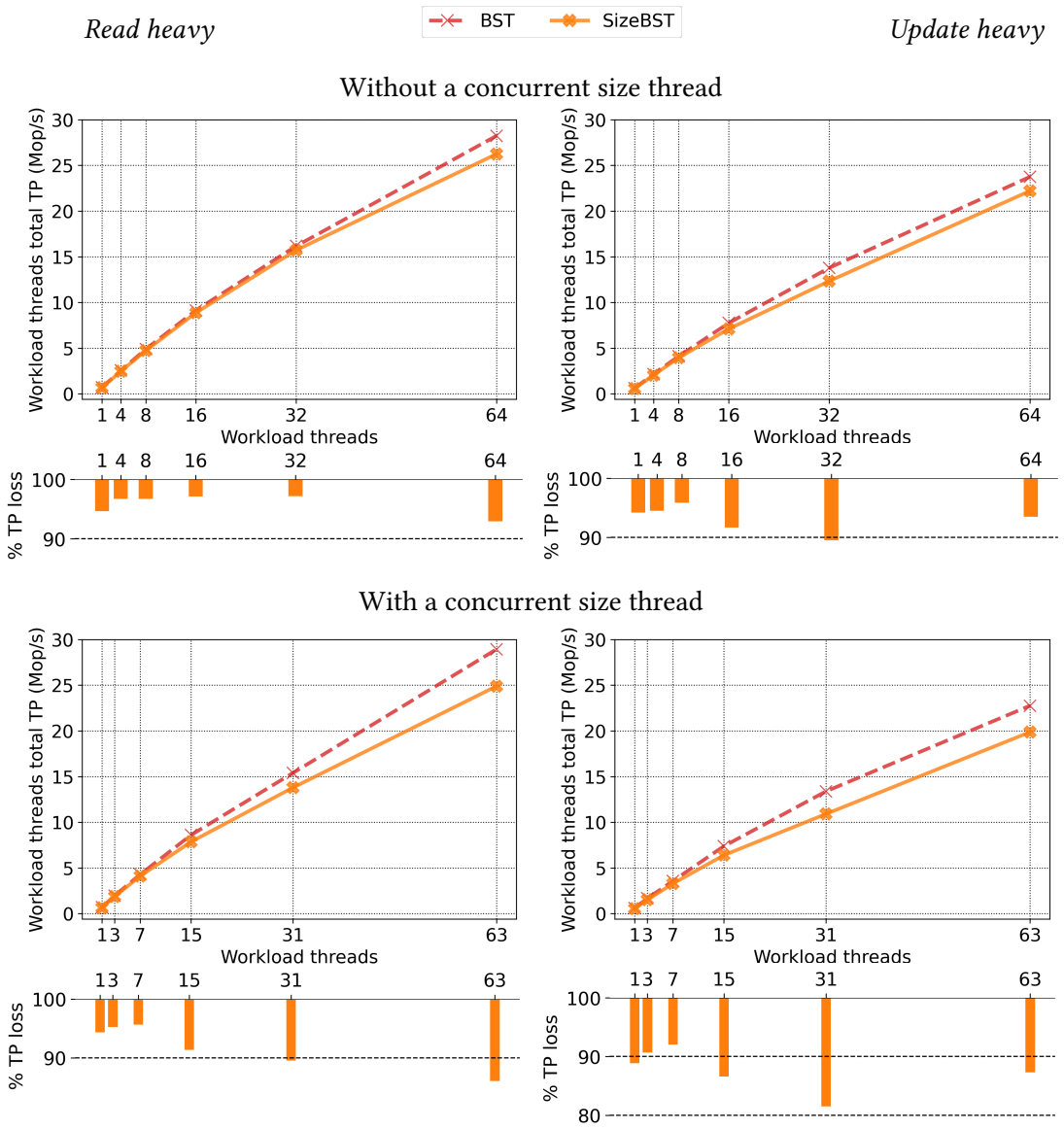


Figure 3.8: Overhead on BST operations

For each experiment, the top graph depicts the number of operations (insert, delete and contains) applied to the data structure per second by the workload threads altogether, measured in million operations per second. The curve of the baseline data structure appears along with the curve of its transformed version with size support. The bottom bar graph shows the throughput of the transformed data structure divided by that of the baseline data structure (in percentages), to demonstrate the throughput loss of the transformed data structure's operations. For instance, 90% signify that the transformed workload threads reach 90% of the throughput of the baseline workload threads. The throughput loss is worse for an update-heavy workload than for a read-heavy workload, and worse when a concurrent size is executed. Still, the relative throughput in all experiments varies in the range of 80% to 99%, i.e., a throughput loss of 1% to 20%. We bring a breakdown of the overhead by operation type in Section 3.9.1.

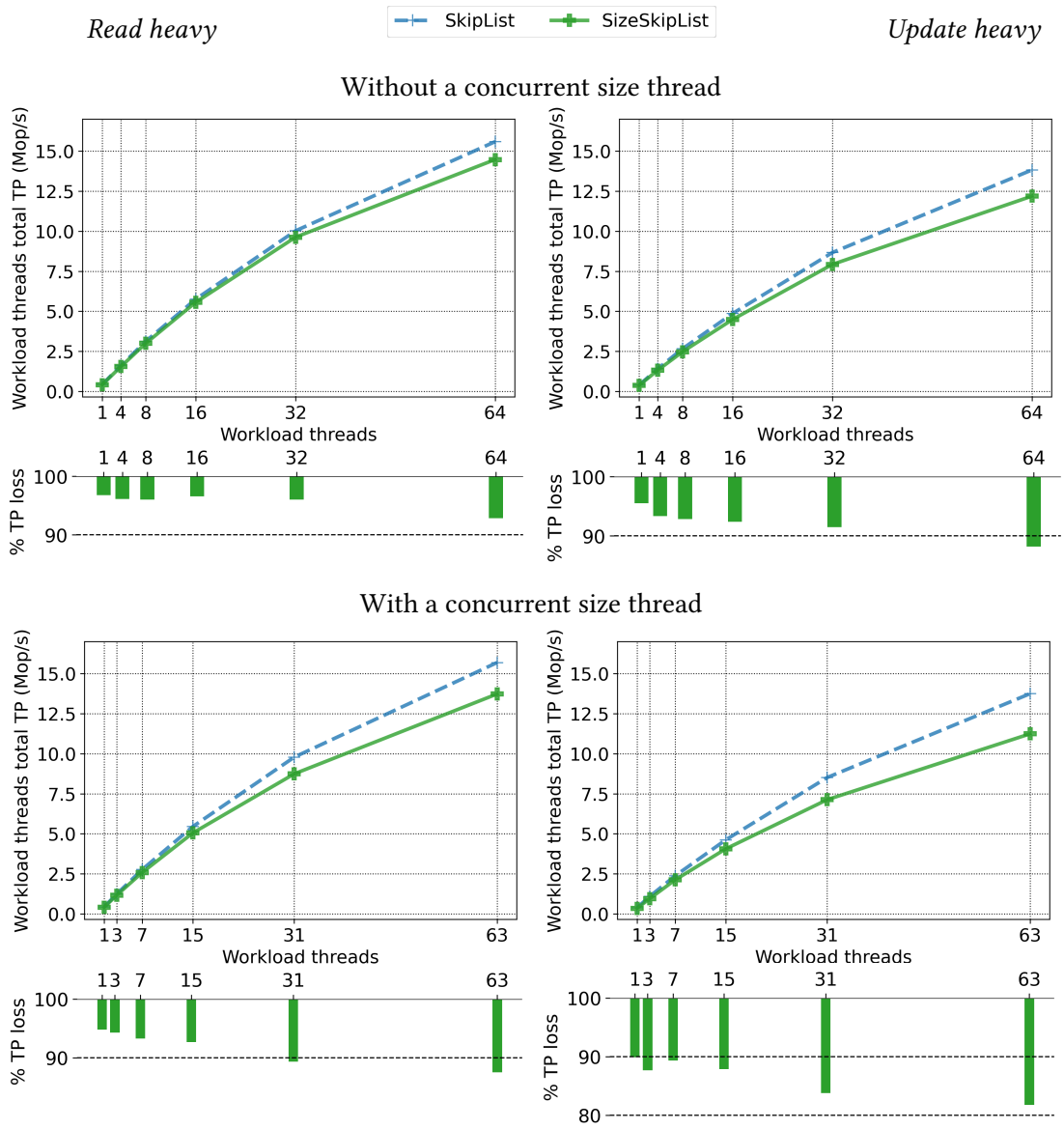


Figure 3.9: Overhead on skip list operations

Varying data-structure size To measure the effect of the number of elements in the data structure on the size throughput, we run experiments on different data-structure sizes, varying between 1M and 100M, with 32 concurrent threads—one size thread and 31 workload threads. Figure 3.10 presents the throughput of the size thread, measured in thousand size operations per second. Each curve shows the size throughput for another transformed data structure, per different initial sizes. The results demonstrate that our size-computation methodology is not sensitive to the data-structure size. This is due to the metadata array, on which the size operates instead of traversing the data structure itself. In contrast, obtaining the size using a snapshot-based method causes performance degradation as the size increases, as shown for VcasBST-64 in Figure 3.11 which presents the corresponding graphs for the competitors. SnapshotSkipList demonstrates a very low size throughput: for a data-structure size of 1M it executes 1.4 size operations per second in average for the read-heavy workload and 1 size

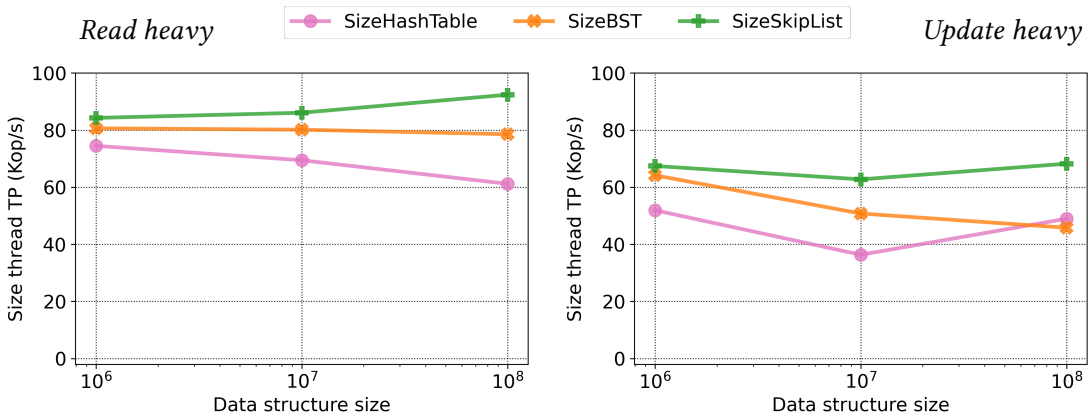


Figure 3.10: Size throughput as a function of data-structure size

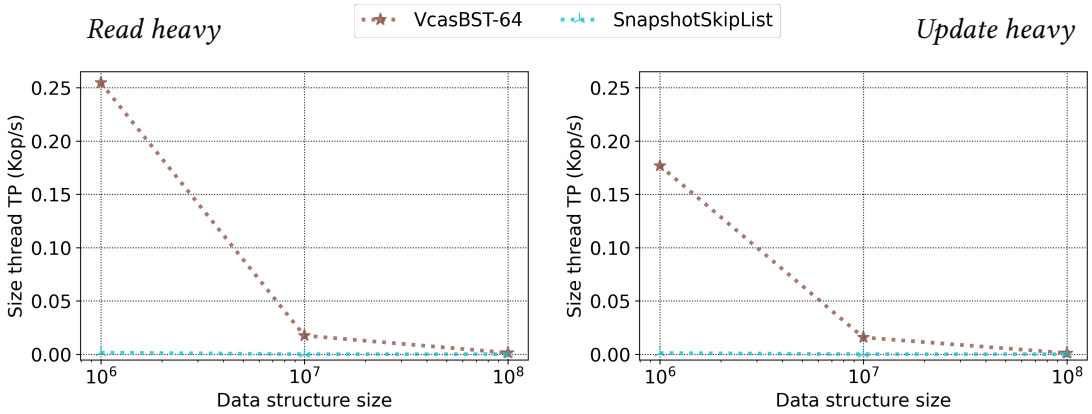


Figure 3.11: Snapshot-based size throughput as a function of data-structure size

operation per second for the update-heavy workload; and for bigger data-structure sizes it executes less than 1 size operation per second.

Scalability To assess the scalability of the size operation, we run s size threads, where s varies between 1 and 16, concurrently with 32 workload threads. Figure 3.12 presents the total throughput of all size threads, measured in thousand size operations per second. It shows results for both our transformed data structures, and the snapshot-supporting data structures which demonstrate inferior performance. For each of our transformed data structures, the throughput improves as number of size threads increases. This demonstrates the scalability of our methodology.

Comparison to snapshot-based size Our transformed data structures yield a much better throughput than the competitors, as demonstrated in Figures 3.10–3.12: SizeSkipList demonstrates in these experiments a throughput at least $54806\times$ the throughput of SnapshotSkipList (in some experiments, not even a single size operation on SnapshotSkipList completed within 5 seconds). The throughput of SizeBST in these experiments is between $83-60423\times$ the throughput of VcasBST-64. The performance gap between our transformed data structures and VcasBST-64 is not as large as the gap from SnapshotSkipList, because

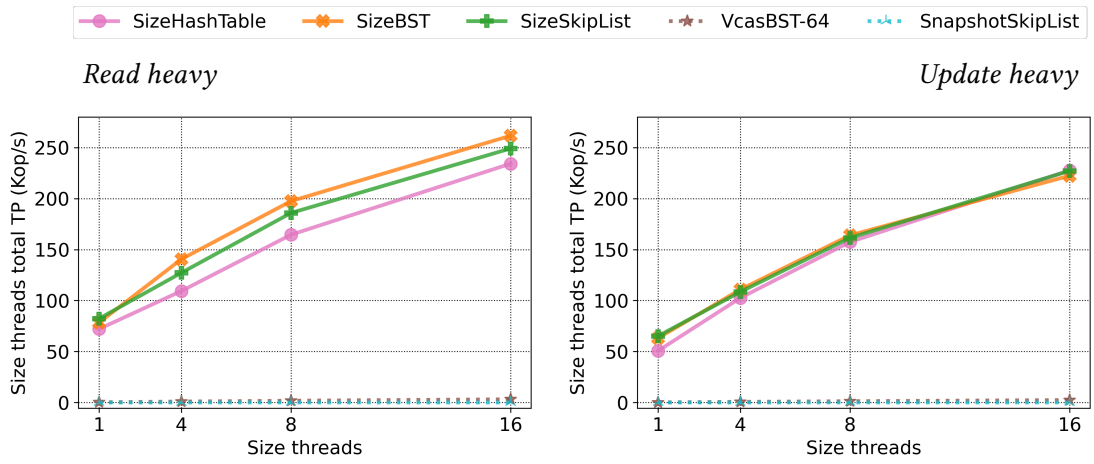


Figure 3.12: Size scalability

VcasBST-64 succeeds to improve snapshot performance in comparison to SnapshotSkipList, but not without a cost—it pays with higher space overhead.

3.9.1 Overhead Breakdown by Operation Type

We performed measurements to assess the overhead breakdown by operation type (insert / delete / contains). Similarly to the above overhead measurements, we compare the performance of workload threads for the transformed data structures versus the baseline data structures. But here, in addition to comparing the combined throughput of all three types of operations by all workload threads (i.e., the total number of all operations divided by the total time they ran), we compare also the total throughput of all workload threads *per operation type* (namely, the total number of insertions by all threads divided by the total time the insertions ran, and the same for deletions and for contains calls). The results appear in Figure 3.13. In most measurements, the throughput loss is highest for insert operations and lowest for contains operations.

3.10 Conclusion

In this work we addressed the problem of obtaining a correct size of a concurrent data structure. We showed that existing solutions in the literature are either inefficient or incorrect (even in a very liberal sense). We then presented a methodology for adding a linearizable size operation to concurrent data structures that implement sets or dictionaries. Our methodology was shown to yield attractive theoretical properties in terms of progress guarantees and asymptotic complexity. Evaluation demonstrated that while incurring some overhead on the data-structure's original operations, the methodology yields a size operation that provides an orders-of-magnitude performance improvement over existing solutions. We additionally illustrated that the size operation is scalable and insensitive to the data-structure size.

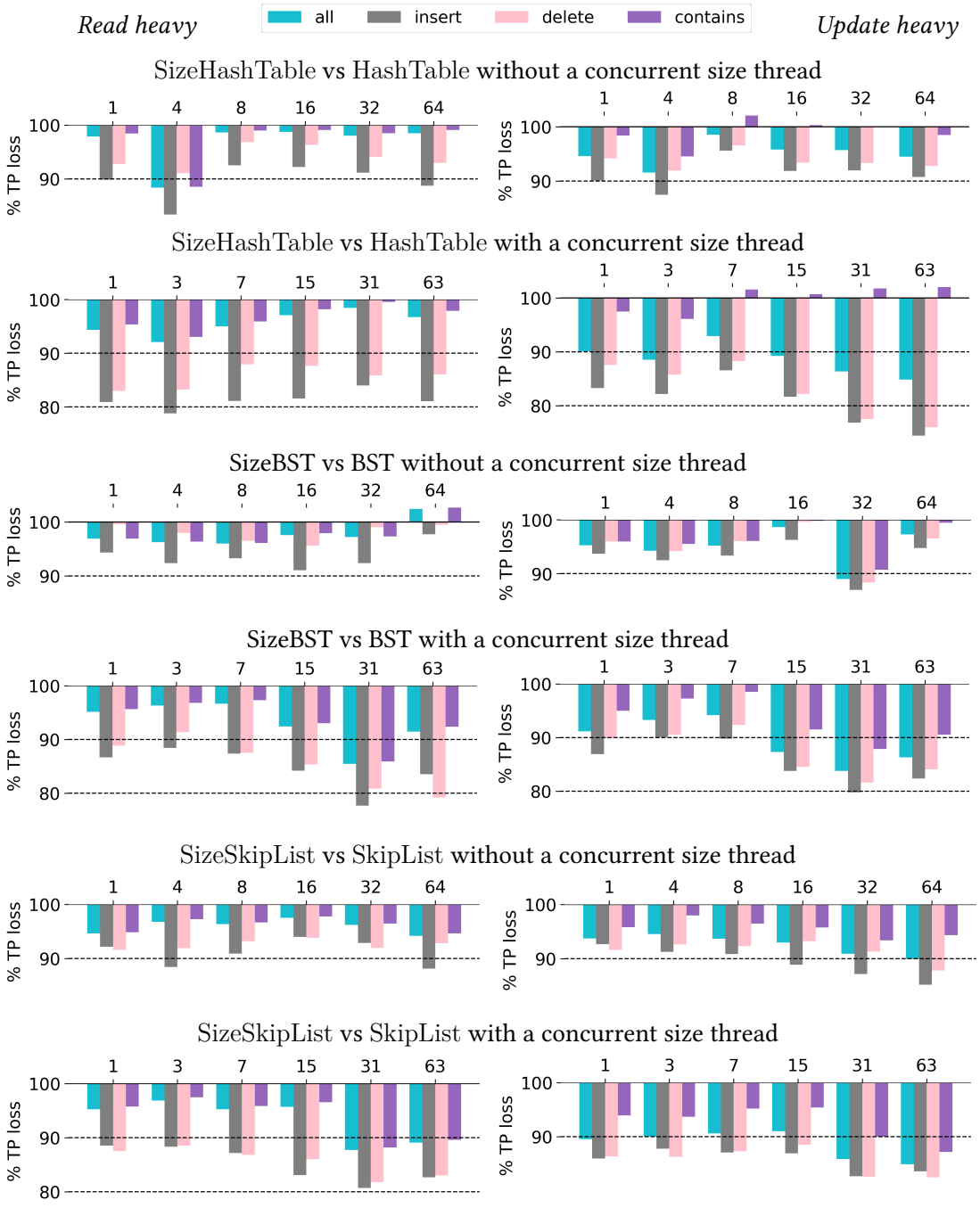


Figure 3.13: Overhead breakdown by operation type

Chapter 4

Linearizability: A Typo

This chapter is based on the work presented at [SHP21a] and [SHP21b].

4.1 Introduction

Linearizability is the prevalent correctness condition for concurrent executions on shared objects. It determines whether a concurrent execution is correct by relating it to a sequential execution that satisfies the sequential specification of the object. To relate a valid sequential execution to a concurrent one, linearizability specifies an order of the concurrent operations, denoted *linearization order*. On the one hand, linearizability requires that if we execute the operations sequentially one by one according to their linearization order (with the same parameters as in the concurrent execution), we obtain a sequential execution (with the same operation results as in the concurrent execution) that satisfies the sequential specification of the object. On the other hand, linearizability dictates that the linearization order preserve the order of non-overlapping operations in the original concurrent execution. Namely, if an operation op_1 completes before another operation op_2 begins, then op_1 must precede op_2 in the linearization order. A concurrent execution is called linearizable if it can be related as above to a legal sequential execution (i.e., one that satisfies the sequential specification of the object). The formal definition is provided in Section 4.2.

So far, we ignored pending invocations in the execution. These are invocations of operations that start during the execution, but do not complete. The issue that we point out in this work concerns the treatment of pending invocations. The original linearizability definition provides a treatment of pending invocations, stating which executions with pending invocations are linearizable. However, in this work we argue that, due to a typo, this treatment of pending invocations is lacking. It contradicts our intuition about linearizable executions, and furthermore, causes the locality and nonblocking properties to not hold. We then propose a simple fix for the typo that fits intuition and obtains these desirable properties also for executions with pending invocations.

Linearizability allows eliding some of the pending invocations, namely, excluding their operations from the related sequential execution. This can be interpreted as operations that

do not yet take effect before the execution ends. The rest of the pending invocations appear in the sequential execution with a *response*, i.e., completion and returned results. These can be interpreted as operations in the concurrent execution that have taken effect although their responses have not yet been returned to the caller. The responses appended in the sequential execution are determined in a way that fits the overall execution. In particular, responses are set so that the related sequential execution satisfies the sequential specification of the object.

The problem that arises, due to the typo in the original definition, is that the definition does not restrict the order of operations that have pending invocations, even when these operations take effect, i.e., are included in the related sequential execution. In particular, a pending invocation of an operation *op* may be placed in a linearization order before other operations that completed earlier in the execution, even operations that completed before *op* started. Imagine an execution that starts with an operation op_1 that reads a shared variable x. While x is initially 0, the operation weirdly reads the value 1, and then completes and returns 1. Later a new operation op_2 is invoked on a different process. Operation op_2 writes 1 into x and does not complete before the execution ends. Intuitively, this does not seem like an acceptable linearizable execution. However, under the existing definition with the typo, it is linearizable, because the pending invocation of operation op_2 (that writes 1) can be ordered before the completed operation op_1 that reads 1.

Interestingly, beyond contradicting intuition, the typo in the original definition does not allow it to yield neither locality nor the nonblocking property. In Section 4.3 we describe the intuitive problem with the typo in the definition and formally show that it is not local neither nonblocking.

We propose a (syntactically very minor) modification to the definition that restricts the linearization order of operations with pending invocations that take effect. Similarly to completed operations, operations with pending invocations are ordered later than any operation that completes before they start. This modification makes the odd execution described above not linearizable. In Section 4.2 we recall the formal original definition of linearizability (with the typo). In Section 4.4 we formally specify the amended definition, and in Section 4.5 we revisit the issues that the typo raises and show that they are resolved. We also show in Section 4.7 that an alternative equivalent definition of linearizability [Lyn96] is actually not equivalent to the definition of linearizability with the typo, but it is equivalent to the version that we propose without the typo. In Section 4.6 we discuss an alternative interpretation of the original definition (with the typo, but with a different definition of what an operation means) and explain why this alternative interpretation is problematic as well. Finally, we put the various linearizability definitions covered throughout this chapter in context in Section 4.8.

It is clear that the flaw in the definition is a typo, not a conceptual error, and the authors' intended meaning is clear in context. We believe no prior paper was rendered incorrect by relying on the original definition. However, linearizability is extremely important for concurrent executions. It has been used in thousands of papers and the definition with the typo has been replicated in numerous subsequent publications. We believe it is important to point out this typo and provide a rigorous discussion and a fix. The issue of pending invocations is be-

coming increasingly important as architectures with non-volatile main memory become commonplace. Non-volatile memory models encompass various definitions [e.g., IMS16; AF03; GL04; BGT15] where a major focus is dealing with invocations pending at the time of a crash. In this realm, pending invocations become critically important, making the fix of this typo timely.

4.2 System Model and Linearizability Definition

The system model was presented in Chapter 2. We recall the exact definition of operations from [HW90], which is inherent to the definition of linearizability:

Definition 4.2.1. *(Operation)* An *operation* in a history is a pair consisting of an invocation and the next matching response.

The original definition of linearizability according to [HW90] follows:

Definition 4.2.2. *(Linearizability)* A well-formed history H is *linearizable* if it has an extension H' such that:

L1: There exists a legal sequential history S, to which $complete(H')$ is equivalent.

L2: $<_H \subseteq <_S$.

S is denoted the *linearization* of H, and operations that appear in *complete*(H') are denoted *linearized* operations. Definition 4.2.2 requires the existence of a linearization S that satisfies two conditions. Condition L1 refers to each process individually, guaranteeing that all its linearized operations are in the same order and with the same results as in the legal sequential history S. Due to this equivalence to a legal sequential history, operations in H act as if they were interleaved at the granularity of complete operations, and adhere to the sequential specification. Condition L2 guarantees that S preserves the order of non-concurrent operations in H, so that it respects possible dependencies between operations in H.

4.3 Issues with the Definition with the Typo

Linearizability enforces real-time precedence order on operations. Definition 4.2.2 enforces it only on operations that include both an invocation and a response in the given history. Fixing the typo extends the enforcement to linearized operations related to pending invocations as well, so that overall, the order is enforced on all linearized operations. To establish the necessity of the typo fix, we present in Section 4.3.1 motivating examples of executions classified as linearizable by Definition 4.2.2 although intuitively they do not seem like acceptable linearizable executions. Moreover, we show in Section 4.3.2 that linearizability as defined in Definition 4.2.2 is not local, and show in Section 4.3.3 that it is not nonblocking.

4.3.1 Executions Counter-Intuitively Classified As Linearizable

We start with two simple examples of executions on a single object, that intuitively seem non-linearizable, but are classified as linearizable by Definition 4.2.2. This stems from allowing operations related to pending invocations to appear to take effect before operations by other processes that precede them, since L2 enforces order among H's operations only, and excludes all pending invocations of H.

Consider the execution H_1 that appears in Figure 4.1, involving two processes: A and B, operating on a register object initialized to 0. H_1 is intuitively unacceptable, as A cannot "predict the future" and read the value that B has not yet even asked to write, and it cannot distinguish between the given execution and an execution in which B does not invoke any write. Therefore, A should return 0 and not 1. Nevertheless, Definition 4.2.2 classifies the execution as linearizable, as there is an extension H_1' and a legal sequential execution S_1 (see Figure 4.1) that adhere to the conditions in Definition 4.2.2: L1 holds since the events per process in $complete(H_1')$ and in S_1 are identical. L2 vacuously holds since it enforces order only on operations of H_1, and H_1 has a single operation (since a pending invocation does not count as an operation, see Definition 4.2.1). In particular, L2 does not force B's write to occur in S_1 after the read operation by A.

We bring as a second example the execution H_2 demonstrated in Figure 4.2, involving two processes: A and B, operating on a FIFO queue initialized to be empty. H_2 is intuitively unacceptable due to the return value of the dequeue operation, which returns an item enqueued by the second enqueue operation rather than the first one, thus violating the FIFO requirement. Nonetheless, as Condition L2 of Definition 4.2.2 does not enforce linearizations of H_2 to place A's enqueue before B's enqueue, H_2 is considered linearizable, by the extension H_2' and the linearization S_2 that appear in Figure 4.2.

Figure 4.1: Executions on a register:

- H_1 – an execution with a return value conflicting with the order between the pending write and the preceding read.
- $complete(H_1')$ – identical to H_1', an extension of H_1.
- S_1 – a linearization of H_1.

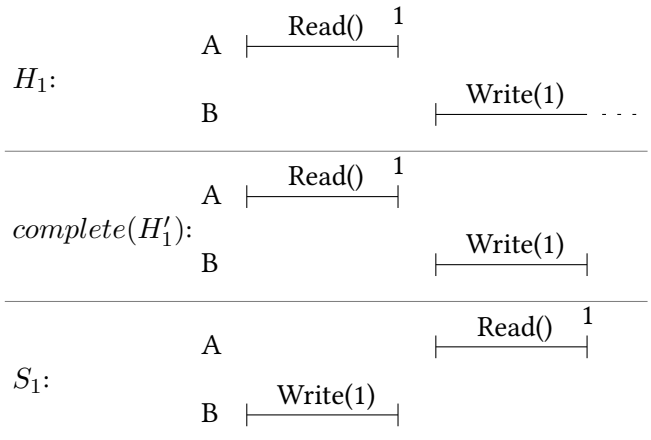


Figure 4.2: Executions on a FIFO queue:

- H_2 – an execution with a return value conflicting with the order between the pending enqueue and the preceding enqueue.
- $complete(H_2')$ – identical to H_2', an extension of H_2.
- S_2 – a linearization of H_2.

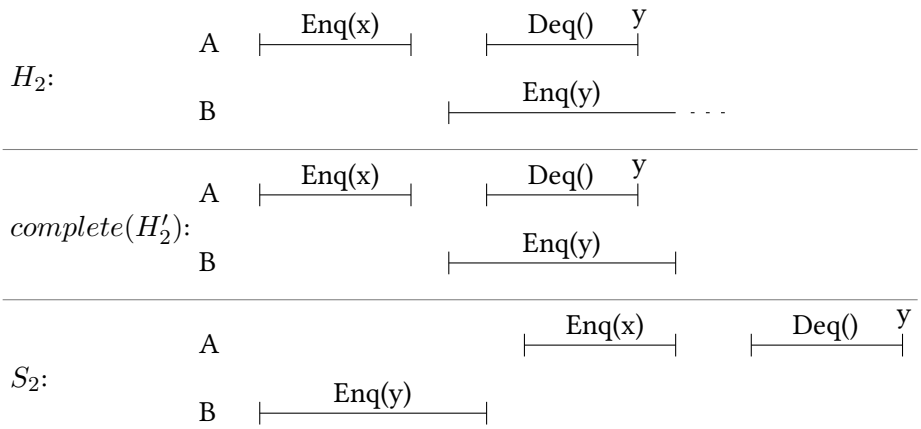
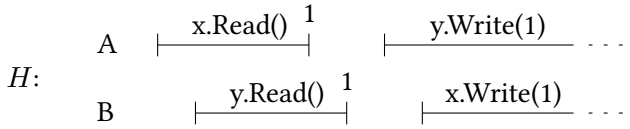


Figure 4.3: H, a non-linearizable execution on two registers, although the object subhistory for each register is linearizable



4.3.2 Linearizability With The Typo Is Not Local

A property of a concurrent system is *local* (synonymously *composable*) if the system as a whole satisfies it whenever each object in the system satisfies it individually. Locality enables implementing and verifying objects independently, thus maintaining modularity. To demonstrate that linearizability as defined with the typo is not local, we bring an execution H in Figure 4.3, involving two processes: A and B, operating on two register objects initialized to 0: x and y. For each of x and y, the object subhistory of the presented execution H is similar to H_1 (see Figure 4.1). As shown in Section 4.3.1, these subhistories are linearizable by Definition 4.2.2. However, H as a whole is not linearizable by Definition 4.2.2: An appropriate extension H' must include responses to both writes for the writes to be included in $complete(H')$, otherwise there will be no legal sequential execution S equivalent to $complete(H')$, because the read operations could not legally return 1. Together with the order enforced by Condition L1 on operations by each process, we get the following order requirements, which form a cycle: x.Read() must occur before y.Write(1) for S to preserve the order of A's events (due to Condition L1 that requires $S|A = complete(H')|A$); y.Write(1) must occur before y.Read() for S to be a legal register history (which dictates in particular that 1 be a legal return value of y.Read()); y.Read() must occur before x.Write(1) for S to preserve the order of B's events (due to Condition L1 that requires $S|B = complete(H')|B$); and finally x.Write(1) must occur before x.Read() for S to be a legal register history.

4.3.3 Linearizability With The Typo Is Not Nonblocking

We look at pending invocations of *total* operations, which are operations defined for every object value (following the terminology of [HW90]). A property of a concurrent system is *nonblocking* if processes invoking total operations are never forced to wait for another pending invocation to complete. Formally, linearizability is nonblocking if for each linearizable execution that has some pending invocation inv of a total operation, there exists a matching response for inv such that appending it to the execution results in a linearizable execution.

To demonstrate that linearizability as defined with the typo is not nonblocking, we look at the execution H_1 (see Figure 4.1). This execution is linearizable by the original definition (as shown above) and has a pending invocation – the invocation of B's write. Appending a response $resp$ for this write to H_1 results in the non-linearizable execution $H_1 \cdot resp$: The pending write becomes an operation in $H_1 \cdot resp$ and is thus ordered by $<_{H_1 \cdot resp}$ as appearing after A's read. Condition L2 of Definition 4.2.2 dictates that a linearization of $H_1 \cdot resp$ respect this order and place the write after the read, which makes it impossible for A's read to legally return 1.

Another counterexample is the execution H_2 (see Figure 4.2). It is linearizable by the original definition (as shown above) and has a pending invocation – the invocation of B's enqueue. Appending a response $resp$ for this enqueue to H_2 results in the non-linearizable execution $H_2 \cdot resp$: The pending enqueue becomes an operation in $H_2 \cdot resp$ and is thus ordered by $<_{H_2 \cdot resp}$ as appearing after A's enqueue. Condition L2 of Definition 4.2.2 dictates that the linearization respect this order and place B's enqueue after A's enqueue, which makes it impossible for A's dequeue to return y while obeying the FIFO specification of a FIFO queue.

4.4 Amended Linearizability

We bring the amended definition of linearizability, which fixes a typo in Condition L2 with a fix that enforces real-time precedence order on linearized operations related to pending invocations:

Definition 4.4.1. *(Amended Linearizability)* A well-formed history H is *linearizable* if it has an extension H' such that:

L1: There exists a legal sequential history S, to which $complete(H')$ is equivalent.

L2: $<_{complete(H')} \subseteq <_S$.

L2 is equivalent to $<_{H'} \subseteq <_S$ because $complete(H')$ and H' differ only in pending invocations and according to the definitions in [HW90], which we use too, pending invocations are not considered operations and a happens-before order does not apply to them. While writing L2 as above makes the definition easier to understand, note that writing L2 as $<_{H'} \subseteq <_S$ provides a fix that is within a single prime sign from the original definition. This missing prime is the typo in the original definition.

Some papers have used an alternative definition of operations, in which pending invocations are also considered as operations, to which a happens-before relation applies. We consider this alternative definition (with the original linearizability definition) in Section 4.6 and show that it does not yield an adequate definition for linearizability.

4.5 Issues Revisited

We explain how the typo fix solves the issues raised in Section 4.3.

4.5.1 Executions Become Non-Linearizable As Expected

We have shown in Section 4.3.1 that H_1 (see Figure 4.1) is linearizable by Definition 4.2.2 although intuitively it is not an acceptable linearizable execution. With the amendment of L2, H_1 becomes non-linearizable by Definition 4.4.1: To satisfy Condition L1, an appropriate extension H_1' must include a response for the write operation, otherwise there will be no legal sequential execution S_1 equivalent to $complete(H_1')$, because the read operation could not legally return 1. In addition, an appropriate linearization S_1 must satisfy the amended L2 Condition, which dictates that the read precede the write in S_1 because it precedes it in $complete(H_1')$. This means the read must return 0 in S_1 for S_1 to be legal, which contradicts Condition L1 that requires S_1 to have the same return values as $complete(H_1')$.

In a similar fashion, H_2 (see Figure 4.2) becomes non-linearizable by Definition 4.4.1 as desired: an appropriate extension H_2' must include a response for Enq(y); thus, Enq(y) must happen in S after Enq(x); and so Deq() must return x in S, in contradiction to L1.

4.5.2 Linearizability Becomes Local

With the typo fix, the history H demonstrated in Figure 4.3 does not stand anymore as a counterexample to the locality of linearizability, since H's object subhistories for x and y are not linearizable by Definition 4.4.1, as explained for H_1 in Section 4.5.1. We will show that linearizability with the typo fix is indeed local.

Next, we repeat the locality proof from the original paper, explain why the proof does not hold for the definition with the typo, and describe an amendment of the proof that makes it correct for Definition 4.4.1. We start by recalling the theorem and its proof as in the original paper. (We note that H mentioned in the theorem is assumed to be a well-formed history, as all histories in [HW90].)

> **Theorem 4.1.** *H is linearizable if and only if, for each object x, $H|x$ is linearizable.*
>
> *Proof* The "only if" part is obvious.
>
> For each x, pick a linearization of $H|x$. Let R_x be the set of responses appended to $H|x$ to construct that linearization, and let $<_x$ be the corresponding linearization order. Let H' be the history constructed by appending to H each response in R_x. We will construct a partial order $<$ on the operations of $complete(H')$ such that:

(1) For each x, $<_x \subseteq <$, and (2) $<_H \subseteq <$. Let S be the sequential history constructed by ordering the operations of $complete(H')$ in any total order that extends $<$. Condition (1) implies that S is legal, hence that L1 is satisfied, and Condition (2) implies that L2 is satisfied.

Let $<$ be the transitive closure of the union of all $<_x$ with $<_H$. It is immediate from the construction that $<$ satisfies Conditions (1) and (2), but it remains to be shown that $<$ is a partial order. We argue by contradiction. If not, then there exists a set of operations $e_1, \ldots, e_n$, such that $e_1 < e_2 < \cdots < e_n$, $e_n < e_1$, and each pair is directly related by some $<_x$ or by $<_H$. Choose a cycle whose length is minimal.

Suppose all operations are associated with the same object x. Since $<_x$ is a total order, there must exist two operations e_{i-1} and e_i such that $e_{i-1} <_H e_i$ and $e_i <_x e_{i-1}$, contradicting the linearizability of x.

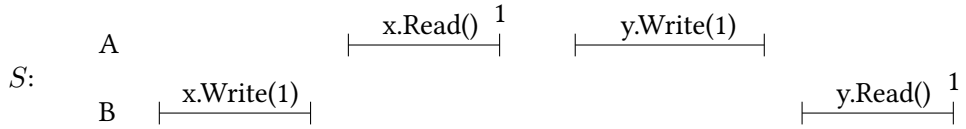
The cycle must therefore include operations of at least two objects. By reindexing if necessary, let e_1 and e_2 be operations of distinct objects. Let x be the object associated with e_1. We claim that none of $e_2, \ldots, e_n$ can be an operation of x. The claim holds for e_2 by construction. Let e_i be the first operation in $e_3, \ldots, e_n$, associated with x. Since e_{i-1} and e_i are unrelated by $<_x$, they must be related by $<_H$; hence the response of e_{i-1} precedes the invocation of e_i. The invocation of e_2 precedes the response of e_{i-1}, since otherwise $e_{i-1} <_H e_2$, yielding the shorter cycle $e_2, \ldots, e_{i-1}$. Finally, the response of e_1 precedes the invocation of e_2, since $e_1 <_H e_2$ by construction. It follows that the response to e_1 precedes the invocation of e_i, hence $e_1 <_H e_i$, yielding the shorter cycle $e_1, e_i, \ldots, e_n$.

Since e_n is not an operation of x, but $e_n < e_1$, it follows that $e_n <_H e_1$. But $e_1 <_H e_2$ by construction, and because $<_H$ is transitive, $e_n <_H e_2$, yielding the shorter cycle $e_2, \ldots, e_n$, the final contradiction.

The above proof does not hold for the definition with the typo since the history S constructed in the proof is not guaranteed to satisfy Condition L1 of linearizability. L1 requires S to preserve the precedence order of the linearized operations by each process. But the constructed S is not guaranteed to preserve the order between two linearized operations by the same process on two different objects, if the later operation of the two is related to a pending invocation in H. For operations on different objects, S is guaranteed to preserve order only among linearized operations whose invocations are not pending in H, due to Condition (2) with the typo in the proof (namely, $<_H \subseteq <$ rather than $<_{complete(H')} \subseteq <$). For example, the history S demonstrated in Figure 4.4 may be constructed by the proof (when using Definition 4.2.2) as a linearization of H from Figure 4.3. Due to the reversed order for B, $complete(H')|B \neq S|B$, and so S does not satisfy L1 for H.

To fix the proof, we replace every reference to $<_H$ in the proof with $<_{complete(H')}$, similarly to the typo fix of the definition. The fixed proof is correct when applied to the lin-

Figure 4.4: S, a sequential history that might be constructed in the proof of Theorem 4.1 as a linearization of H



earizability definition with the typo fix. In particular, thanks to the fix of Condition (2), L1 is guaranteed to be satisfied. We note that the fixed proof naturally holds for the definition with the typo fixed, but it does not work for the linearizability definition with the typo, which is not local as proven above. The barrier in this case is that the constructed $<$ is not necessarily a partial order, but rather might contain cycles. In detail, the above proof argues by contradiction that $<$ is a partial order. It assumes an operation cycle $e_1 < e_2 < \cdots < e_n < e_n < e_1$ satisfying certain properties and needs to reach a contradiction. In the case that all operations in this cycle are associated with the same object x, then as the proof states, there must exist two operations in the cycle, e_{i-1} and e_i, such that $e_{i-1} <_{complete(H')} e_i$ (referring to $<_{complete(H')}$ rather than $<_H$ is due to the proof fix) and $e_i <_x e_{i-1}$. This contradicts the linearizability of $H|x$ if linearizability is defined with the typo fix, namely, L2 for $H|x$ requires $<_{complete(H')|x} \subseteq <_x$, because then $e_{i-1} <_{complete(H')} e_i$ leads to $e_{i-1} <_x e_i$, contradicting the asymmetry of $<_x$. However, without the definition fix (namely, with L2 for $H|x$ requiring $<_{H|x} \subseteq <_x$), we do not reach a contradiction if e_i is an operation related to a pending invocation of H. The reason is that for such an e_i, $e_{i-1} <_{complete(H')} e_i$ does not lead to $e_{i-1} <_{H|x} e_i$ (since $<_{H|x}$ does not refer to e_i at all), and so L2 does not imply $e_{i-1} <_x e_i$. Consider, for example, H from Figure 4.3, and let e_1 be x.Read() and e_2 be x.Write(1). $<$ constructed in the proof contains the cycle $e_1 < e_2 < e_1$, which stems from $e_1 <_{complete(H')} e_2$ and $e_2 <_x e_1$.

4.5.3 Linearizability Becomes Nonblocking

With the typo fix, the histories H_1 and H_2 do not stand anymore as counterexamples to linearizability being nonblocking, since they are not linearizable by Definition 4.4.1, as explained in Section 4.5.1. We will show that linearizability with the typo fix is indeed nonblocking.

Next, we repeat the nonblocking property proof from the original paper, explain why the proof does not hold for the definition with the typo, and why it does hold for Definition 4.4.1. We start by recalling the theorem and its proof as in the original paper.

> **Theorem 4.2.** *Let inv be an invocation of a total operation. If $\langle x\ inv\ P\rangle$ is a pending invocation in a linearizable history H, then there exists a response $\langle x\ res\ P\rangle$ such that $H \cdot \langle x\ res\ P\rangle$ is linearizable.*
>
> *Proof* Let S be any linearization of H. If S includes a response $\langle x\ res\ P\rangle$ to $\langle x\ inv\ P\rangle$, we are done, since S is also a linearization of $H \cdot \langle x\ res\ P\rangle$. Otherwise, $\langle x\ inv\ P\rangle$ does not appear in S either, since linearizations, by definition, include no pending invocations. Because the operation is total, there exists a response $\langle x\ res\ P\rangle$ such that $S' = S \cdot \langle x\ inv\ P\rangle \cdot \langle x\ res\ P\rangle$ is legal. S', however, is a linearization of $H \cdot \langle x\ res\ P\rangle$, and hence is also a linearization of H. ■

The first part of the above proof does not hold for the definition with the typo, as Condition L2 of Definition 4.2.2, applied to the history $H \cdot \langle x\ res\ P \rangle$, might not be satisfied by S: S, as a linearization of H, is guaranteed by L2 of Definition 4.2.2 to respect the precedence order $<_H$. The issue is that $<_H$ does not enforce any order on pending invocations of H, in particular $\langle x\ inv\ P \rangle$. Since this invocation is not pending in the history $H \cdot \langle x\ res\ P \rangle$, L2 for this history requires more than L2 for H guarantees. In particular, it requires that a linearization of $H \cdot \langle x\ res\ P \rangle$ order the operation of $\langle x\ inv\ P \rangle$ after any operation completed beforehand. However, S might not satisfy it (like in the examples brought in Section 4.3.3).

The first part of the proof does hold for the amended definition: If S is a linearization of H by Definition 4.4.1 that includes a response $\langle x\ res\ P \rangle$ to $\langle x\ inv\ P \rangle$, then there exists an extension H' of H, which includes $\langle x\ res\ P \rangle$ as the first response appended after H, such that S and H' satisfy Definition 4.4.1 for H. This H' is also an extension of $H \cdot \langle x\ res\ P \rangle$, and so the same S and H' satisfy Conditions L1 and L2 of Definition 4.4.1 for $H \cdot \langle x\ res\ P \rangle$ as well.

4.6 An Alternative Interpretation

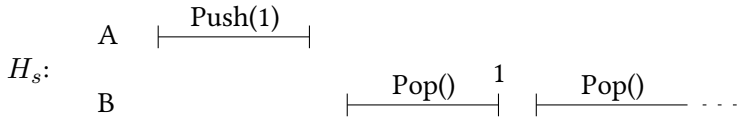
While the original paper only considers completed operations, i.e., an operation is a pair of an invocation and the next matching response, it might seem that if we also consider pending invocations as valid operations, as some papers do, then the original definition may work adequately, leading to an easy fix for the typo in the original definition. In this section we show that this is not the case. Formally, an alternative definition for an operation would be:

Definition 4.6.1. *(Operation - alternative definition)* An *operation* in a history is either a pair consisting of an invocation and the next matching response, or a pending invocation only in case of an invocation that has no matching response in the execution.

This leads to an alternative interpretation of the linearizability definition with the typo, with $<_H$ applied to operations by Definition 4.6.1 – including pending invocations. This interpretation of operations does solve the problem pointed out in Section 4.3 for the original linearizability definition, since it makes Condition L2 cover pending invocations. But this brings about a new problem, as not only linearized pending invocations are covered, but also pending invocations that are not linearized, namely, eliminated from $complete(H')$ and S (which are equivalent by Condition L1). The definition by this interpretation might exclude an execution that seems legitimately linearizable: for an execution containing a pending invocation e_2 that cannot be legally linearized (and hence cannot appear in a linearization S) and an operation e_1 that precedes e_2, there exists no appropriate linearization S, since L2 implies that $e_1 <_S e_2$ and in particular that e_2 appears in S (including a response, as S is equivalent to $complete(H')$ by L1, which means it contains no pending invocations).

An example of such a history that seems naturally linearizable but would be ruled out by the original definition with the alternative interpretation of an operation follows. Consider a stack object, where if the stack is empty, then a popping process spins until an item is pushed

Figure 4.5: H_s, an execution on a stack with a second pop that cannot be completed



into the stack. Consider the history H_s with processes A and B illustrated in Figure 4.5. The first pop's response precedes the second pop's invocation in H_s, hence $1^{st}\ Pop <_{H_s} 2^{nd}\ Pop$ (interpreting the happens-before relation relying on the alternative operation definition). Condition L2 of the linearizability definition implies that the same relation appears in S: $1^{st}\ Pop <_S 2^{nd}\ Pop$, and in particular the second pop appears in S. However, the second pop cannot be included in S because it cannot be legally completed on an empty stack.

This problem regarding the definition of linearizability in the alternative interpretation is different from the problem with the original interpretation (pointed out in Section 4.3), but we remark that applying our fix, i.e., changing L2 to $<_{complete(H')} \subseteq <_S$, solves this problem as well and can make the alternative interpretation be an adequate (equivalent) definition for linearizability.

4.7 An Equivalent Definition

According to the intuitive discussion in the original paper [HW90], linearizability provides the illusion that each operation takes effect instantaneously at some point between its invocation and its response. This point was later denoted a *linearization point*. Referring to the intuitive meaning of linearizability, it makes sense that if a pending invocation takes effect, it does so instantaneously at some point after the invocation and before the end of the execution. Such an interpretation of linearizability has appeared in [Lyn96]. Moreover, many data structure implementations [e.g., MS96; Mic02] are proven to be linearizable by listing linearization points as locations in the code for each of their methods. These locations naturally occur during the method call, after the invocation (and before the response or the end of the execution), obliviously of whether the invocation has a matching response in the original execution.

We next specify an equivalent definition of linearizability, similar to the *atomicity* definition in [Lyn96], which formalizes the above intuitive interpretation of linearizability.

Definition 4.7.1. *(Linearizability by Linearization Points)* A well-formed history H is *linearizable* if there exist distinct points in H, denoted linearization points, satisfying the following:

1. For each operation, there exists a linearization point between its invocation and its response.

2. There exists a subset T of H's pending invocations, such that for each invocation inv in T there exists a linearization point after the invocation, and there exists a response denoted $resp_{inv}$ for the invocation.

Such that if we place each invocation that has a matching response and its matching response one right after another at their respective linearization point, do the same for each invocation inv in T and its response $resp_{inv}$, and exclude pending invocations not in T, then the resulting sequence of invocations and responses, denoted S, is a legal sequential history.

The original linearizability definition with the typo is not equivalent to Definition 4.7.1. As a counterexample, H_1 demonstrated in Figure 4.1 is linearizable by Definition 4.2.2 as shown in Section 4.3.1, but not by Definition 4.7.1, since the pending write must be linearized for the read to return 1, but placing the write's linearization point after the read's one cannot yield a legal register sequential history.

We next show that fixing the typo in the linearizability definition makes the definition equivalent to Definition 4.7.1, which formalizes the intuition behind linearizability.

Claim 4.7.2. *Definition 4.7.1 (linearizability by linearization points) is equivalent to Definition 4.4.1 (amended linearizability).*

Proof First we prove that Definition 4.7.1 implies Definition 4.4.1: Assume H is linearizable by Definition 4.7.1. Let T and S be a subset and a history that satisfy the definition. We will prove that H is linearizable by Definition 4.4.1 with the same S. Form H' from H by appending (in some arbitrary order) for each invocation in T, the response appended for it in S. L1 holds: For each process, $complete(H')$ is made of the same events as S. Their order is the same in $complete(H')$ and S, since S is constructed by "shrinking" each operation (invocation and response) to its linearization point, which is placed by Definition 4.7.1 between an invocation and a following response. L2 holds as well, since if one operation precedes another operation in $complete(H')$, meaning the first operation's response happens before the second operation's invocation in $complete(H')$, then this order is preserved in S, in which the response of the first operation is moved to an earlier point (to the linearization point of the first operation) and the invocation of the second operation is moved to a later point (to the linearization point of the second operation).

We proceed to prove the other direction - Definition 4.4.1 implies Definition 4.7.1. Assume H is linearizable by Definition 4.4.1. Let H' and S be an extension of H and a linearization that satisfy the definition. We define T to be the set of all pending invocations in H for which a response is added in H'. Next, we pick linearization points for operations of H and for pending invocations of T, namely for operations of $complete(H')$, which are – due to L1 – the operations of S. Denote the i^{th} operation in S by e_i. We pick the linearization point of e_i to be right after the later of the following: e_i's invocation, and the linearization point of e_{i-1}.

We will prove that our selected linearization points satisfy the requirements of Definition 4.7.1. First, Conditions (1) and (2) of Definition 4.7.1 hold as the linearization point of each e_i is after its invocation in $complete(H')$ by definition, and also before its response as we next show. If not, it implies that the linearization point of e_i is set right after the linearization point of e_{i-1}, and that the linearization point of e_{i-1} is after e_i's response. If the linearization point of e_{i-1} was picked to be right after its invocation, it means that $e_i <_{complete(H')} e_{i-1}$

and we reach a contradiction by Condition L2 of Definition 4.4.1 (thanks to the typo fix). Else, it was picked to be right after the linearization point of e_{i-2}, and we continue with the same arguments until reaching a contradiction (the process is guaranteed to stop at e_1 at the latest). Second, the linearization point order preserves the order of operations in S by our definition of the linearization points. Therefore, the history constructed in Definition 4.7.1 by "shrinking" each operation of S (namely, moving its invocation and response) to its linearization point, is equal to S and is thus a legal sequential history.

4.8 Comparison of all Definition Versions

Let H be a history. Then for each choice of an extension H', we may divide the invocations in H into three categories:

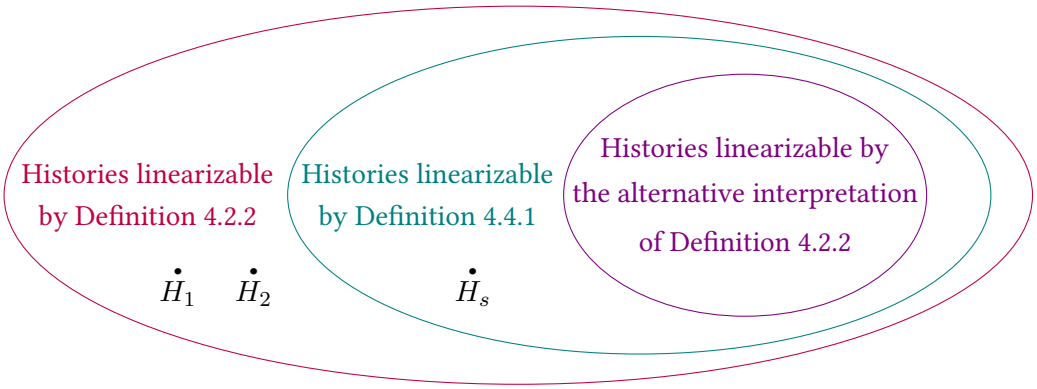
1. Invocations that have a matching response in H.
2. Pending invocations in H that have a matching response in H'.
3. Pending invocations in H that do not have a matching response in H'.

The operations related to invocations of categories (1) and (2) form $complete(H')$. Therefore, as implied by Condition L1 of linearizability, if H is linearizable then the linearization of H which stems from this H' is made of these operations.

The different versions of the linearizability definition differ in the operations among which their L2 Condition enforces order preservation. Definition 4.2.2 forces preserving order among operations whose invocations are of category (1) only. Definition 4.4.1 (and thus the equivalent Definition 4.7.1 as well) dictates that the linearization preserve the order of all operations in $complete(H')$, namely, all linearized operations, which are operations whose invocations are of categories (1) and (2). Since L2 with the typo fix enforces precedence order on additional operations in comparison to the original L2, then the typo fix strengthens the definition of linearizability. Namely, the amended definition eliminates some executions that are linearizable according to the definition with the typo, e.g., executions H_1 (see Figure 4.1) and H_2 (see Figure 4.2). The typo fix does not include any execution not classified as linearizable by the original definition: each execution linearizable by Definition 4.4.1 is linearizable by Definition 4.2.2 as well because $<_H \subseteq <_{complete(H')}$.

Definition 4.2.2 with the alternative interpretation of an operation as described in Section 4.6, forces preserving order among operations whose invocations are of any of the 3 above-mentioned categories, namely, operations related to all invocations. L2 in this interpretation implies that a linearization S includes all pending invocations of H (that are preceded by any response), including those of category (3) – which cannot appear in S by Condition L1. Consequently, for a history to be classified as linearizable, there must exist an extension H' for which H has no invocations of category (3), namely, all pending invocations are linearized (again, referring only to invocations preceded by a response). Hence, the definition in this

Figure 4.6: The relationship between histories categorized as linearizable by the different versions of linearizability



interpretation excludes histories with pending invocations (preceded by some response) that cannot be linearized, like H_s (see Figure 4.5). It is thus stronger than Definition 4.4.1 – in fact, too strong.

The inclusion relations between the histories categorized as linearizable by the different versions of linearizability are illustrated in Figure 4.6.

Chapter 5

The FIDS Theorems: Tensions between Multinode and Multicore Performance in Transactional Systems

This chapter is based on the work presented at [BSS23a] and [BSS23b].

5.1 Introduction

Transactional systems offer a clean abstraction for programmers to write concurrent code without worrying about synchronization issues. This has made them extremely popular and well studied in the last couple of decades [AF15; FTA14; GK08; PPR+15; SWL+20; YPSD16; ZSS+15; TZK+13a].

Many transactional systems in practice are *distributed* across multiple machines [CDE+12; ZXS+21; LM10], allowing them to have many benefits that elude single-machine designs. For example, distributed solutions can scale to much larger data sets, handle much larger workloads, service clients that are physically far apart, and tolerate server failures. It is therefore unsurprising that distributed transactional systems have garnered a lot of attention in the literature, with many designs aimed at optimizing their performance in various ways: minimizing network round trips to commit transactions [KPF+13; SWL+20; ZSS+15; MNLL16; CL12; LMP17], increasing robustness and availability when server failures occur [KPF+13; ZSS+15; SWL+20], and scaling to heavier workloads [ZXS+21; EGG+22].

Due to increased bandwidth on modern networks, new considerations must be taken into account to keep improving the performance of distributed transactional systems. In particular, while traditional network communication costs formed the main bottleneck for many applications, sequential processing within each node is now no longer enough to handle the throughput that modern networks can deliver (through e.g., high-bandwidth links, multicore

NICs, RDMA, kernel bypassing). Thus, to keep up with the capabilities of modern hardware, distributed transactional systems must make use of the parallelism available on each server that they use. That is, they must be designed while optimizing *both* network communication *and* multicore scalability.

Two main approaches have been employed by transactional storage systems to take advantage of the multicore architecture of their servers [Sto85]: *shared-nothing* or *shared-memory*. The shared-nothing approach, where each core can access a distinct partition of the database and only communicates with other cores through message passing, has a significant drawback: cores responsible for hot data items become a throughput bottleneck while other cores are underutilized. To be able to adapt to workloads that stress a few hot data items, the shared memory approach, where each core can access any part of the memory, can be used. However, shared memory must be designed with care, as synchronization overheads can hinder scalability. Fortunately, decades of work has studied how to scale transactional systems in a multicore shared-memory setup [AHM11; AH12; AF15; BHG86; BDFG14; Pap79; PPR+15; AS08]. Thus, there is a lot of knowledge to draw from when designing distributed transactional systems that also employ parallelism within each server via the shared-memory approach.

In this work, we study these systems, which we call parallel distributed transactional systems (PDTSs). Our main contribution is to show that there is an inherent tension between properties known to improve performance in distributed settings and those known to improve performance in parallel settings. To show this result, we first formalize a model that combines both shared memory and message passing systems. While such a model has been formulated in the past [ABC+18], it has not been formulated in the context of transactional systems.

We then describe and formalize three properties of distributed transactional systems that improve their performance. These properties have all appeared in various forms intuitively in the literature [ZSS+15; KPF+13; SWL+20], but have never been formalized until now. We believe that each of them may be of independent interest, as they capture notions that apply to many existing systems. In particular, we first present *distributed disjoint-access parallelism*, a property inspired by its counterpart for multicore systems, but which captures scalability across different distributed shards of data. Then, we describe a property that intuitively requires a *fast path* for transactions: transactions must terminate quickly in executions in which they do not encounter asynchrony, failures, or conflicts. While many fast-path properties have been formulated in the literature for consensus algorithms, transactions are more complex since different transactions may require a different number of network round trips, or message delays, in order to even know what data they should access. We capture this variability in a property we name *fast decision*, intuitively requiring that once the data set of a transaction is known, it must reach a decision within one network round trip. Finally, we present a property called *seamless fault tolerance*, which requires an algorithm to be able to tolerate some failures without affecting the latency of ongoing transactions. This has been the goal of many recent works which focus on robustness and high availability [SWL+20; MAK12; MNLL16; KPF+13; ZSS+15].

Equipped with these properties, we then show the inherent tension that exists between

them and the well-known multicore properties of disjoint-access parallelism and invisible reads, both of which intuitively improve cache coherence and have been shown to increase scalability in transactional systems [RHH09; FFMR10]. More precisely, we present the **Fast decision, Invisible reads, distributed Disjoint-access parallelism, and Serializability (FIDS)** theorem for *sharded* PDTSs: a PDTS that guarantees a minimal progress condition and shards data across multiple nodes cannot simultaneously provide **F**ast decision, **I**nvisible reads, distributed **D**isjoint-access parallelism, and **S**erializability. An important implication of this result is that serializable shared-memory sharded PDTSs that want to provide multicore scalability cannot simply use a two-phase atomic commitment protocol (such as the popular two-phase commit). Furthermore, we turn our attention to *replicated* PDTSs. We discover that a similar tension exists for PDTSs that utilize *client-driven* replication. With client-driven replication replicas do not need to communicate with each other to process transactions. It is commonly used in conjunction with a leaderless replication algorithm to save two message delays [KPF+13; ZSS+15; SWL+20; MNLL16], as well as in RDMA-based PDTSs which try to bypass the replicas' CPUs [DNN+15; SRN+19]. We present a *robust* version of the FIDS theorem, which we call the **Robust Fast decision, Invisible reads, Disjoint-access parallelism, and Serializability (R-FIDS)** theorem: a PDTS (that may or may not shard its data) and utilizes client-driven replication cannot simultaneously provide **R**obustness to failures in the form of seamless fault tolerance, **F**ast decision, **I**nvisible reads, **D**isjoint-access parallelism, and **S**erializability.

Interestingly, similar impossibility proofs appear in the literature, often showing properties of parallel transactional systems that cannot be simultaneously achieved [PPR+15; AHM11; BDFG14]. Indeed, some works have specifically considered disjoint-access parallelism and invisible reads, and shown that they cannot be achieved simultaneously with strong progress conditions [AHM11; PPR+15]. However, several systems achieve both disjoint-access parallelism and invisible reads with weak progress conditions such as the one we require [YPSD16; TZK+13a; DNN+15]. To the best of our knowledge, the two versions of the FIDS theorem are the first to relate multicore scalability properties to multinode scalability ones.

Finally, we show that the FIDS theorems are minimal in the sense that giving up any one of these properties does allow for implementations that satisfy the rest.

In summary, our contributions are as follows.

- We present a transactional model that combines the distributed and parallel settings.
- We formalize three distributed performance properties that have appeared in intuitive forms in the literature.
- We present the FIDS and R-FIDS theorems for parallel distributed transactional systems, showing that there are inherent tensions between multicore and multinode scalability properties.
- We show that giving up any one of the properties in the theorems does allow designing implementations that satisfy the rest.

The rest of this chapter is organized as follows. Section 5.2 presents the model and some preliminary notions. In Section 5.3, we define the properties of distributed transactional systems that we focus on. We present our impossibility results in Section 5.4, and then in Section 5.5, we show that it is possible to build a PDTS that sacrifices any one of the properties. Finally, we discuss related works in Section 5.6 and future research directions in Section 5.7.

5.2 Model and Preliminaries

Communication. We consider a *message-passing* model among n *nodes* (server hosts) and any number of *client processes*, as illustrated in Figure 5.1. Each node has P *node processes*. Messages are sent either between two nodes or between clients and nodes. We consider *partial synchrony* [DLS88]; messages can be arbitrarily delayed until an a priori unknown *GST*, after which all messages reach their target within a known delay Δ. An execution is said to be *synchronous* if GST is at the beginning of the execution. Furthermore, node processes within a single node communicate with each other via *shared memory*. That is, they access *shared base objects* through *primitive operations*, which are atomic operations, such as read, write, read-modify-write (compare-and-swap, test-and-set, fetch-and-increment, etc.), defined in the usual way. A primitive operation is said to be *non-trivial* if it may modify the object. Two primitive operations *contend* if they access the same object and at least one of them is non-trivial. The order of accesses of processes to memory is governed by a *fair scheduler* which ensures that all processes take steps.

Transactions. We consider a database composed of a set of *data items*, Σ, which can be accessed by *read* and *write* operations. Each node N_i holds some subset $\Sigma_i \subseteq \Sigma$, which may overlap with the subsets held on other nodes. A *transaction* T is a program that executes read and write operations on a subset of the data items, called its *data set*, $D_T \subseteq \Sigma$. A transaction T's *write set*, $W_T \subseteq D_T$, and *read set*, $R_T \subseteq D_T$, are the sets of data items that it writes and reads, respectively. Two transactions are said to *conflict* if their data sets intersect at an item that is in the write set of at least one of them.

Transaction Interface. An *application* may execute a transaction T by calling an *invokeTxn(T)* procedure. The invokeTxn(T) procedure returns with a *commit* or *abort* value indicating whether it committed or aborted, as well as the full read and write sets of T, with the order of execution of the operations (relative to each other), and with the read and written values. We say that a transaction is *decided* when invokeTxn(T) returns.

Failure Model. Nodes can fail by crashing; if a node crashes then *all* processes on the node crash as well. We do not consider failures where individual processes crash and we assume clients do not fail. We denote by *failure-free* execution an execution without node crashes.

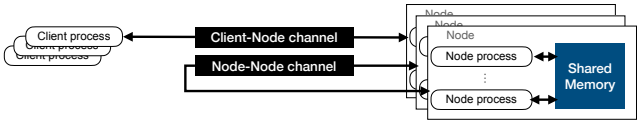


Figure 5.1: Communication mediums between the different types of processes considered in our model.

Handlers and Implementations. An *implementation* of a PDTS provides data representation for transactions and data items, and algorithms for two types of handlers: the *coordinator handler* and the *message handler*. Each handler is associated with a transaction and is executed by a single process. Each process executes at most one handler at any given time, and is otherwise *idle*. The coordinator handler of a transaction T is the first handler associated with T and is triggered by an invokeTxn(T) call on some client process.

The execution of a handler involves a sequence of *handler steps*, which are of one of three types: (1) an *invocation* or *response* step, which is the first or last step of the handler respectively, (2) a primitive operation on a base object in shared memory, including its return value, and (3) sending or receiving a message, denoted send(T, m) or receive(T, m). Each handler step is associated with the corresponding transaction and the process that runs it. The return value in a response step of a transaction's coordinator handler is the return value of invokeTxn described above, and a message handler has no return value.

Executions. An *execution* of a PDTS implementation is a sequence of handler steps and *node crash steps*. Each node crash step is associated with a node. After a node crash step associated with node N_i in execution E, no process on node N_i takes any steps in E. An execution can interleave handler steps associated with different transactions and processes. An *extension* E' of E is an execution that has E as its prefix.

We say that a transaction T's *interval* in an execution E begins at the invocation step of T's coordinator handler, and ends when there are no sends associated with T that have not been received whose target node has not crashed, and all handlers associated with T have reached their response step. Note that the end of a transaction's interval must therefore be a response step of some handler associated with T, but might not be the response step of T's coordinator handler (which may terminate earlier than some other handlers of T). We say that two transactions are *concurrent* in E if their intervals overlap. We say that two transactions, T_1 and T_2, *contend on node* N_i in E if they are concurrent, and there is at least one primitive operation step on node N_i in E associated with T_1 that contends with a primitive operation step in E associated with T_2. We say that T_1 and T_2 *contend* in E if there is some node N_i such that they contend on node N_i in E.

The *projection* of an execution E on a process p, denoted $E|p$, is the subexecution of E that includes exactly all of the steps associated with p in E. Two executions E and E' are *indistinguishable* to a process p if the projections of E and E' on p are identical (i.e., if $E|p = E'|p$).

It is also useful to discuss *knowledge* of properties during an execution. The notion of knowledge has been extensively used in other works [HM90; FHMV04]. Formally, a process p *knows* a property P in an execution E of a PDTS implementation I, if there is no execution E' of I that is indistinguishable to p from E in which P is not true.

We adopt two concepts introduced by Lamport [Lam06b; Lam78] to aid reasoning about distributed systems: *depth of a step*, and the *happened-before* relation. The *depth of a step* s associated with transaction T in execution E is 0 if s is the invocation of T's coordinator

handler. Otherwise, it equals the maximum of (i) the depths of all steps that are before s in E within the same handler as s, and (ii) if s is a receive(T, m) step of a message sent in a send(T, m) step, s', then 1 plus the depth of s'. *Happened-before* is the smallest relation on the set of steps of an execution E satisfying the following three conditions: 1) if a and b are steps of the same handler and a comes before b in E, then a *happened-before* b; 2) if a is a send(T, m) step and b is a receive(T, m) step, then a *happened-before* b; 3) if a *happened-before* b and b *happened-before* c, then a *happened-before* c.

Serializability. Intuitively, a transactional system is *serializable* if transactions appear to have executed in some serial order [Pap79].

Formally, a *committed history* is a higher-level model of an execution defined as a sequence of read and write events of the transactions that committed. A committed history H is *derived* from an execution E if it consists of exactly the following events: for each committed transaction T in E, H includes read and write events associated with T for all the reads and writes in the response step of T's coordinator handler, including the read or written value, in the order stated in the response step's return value (which is the return value of the call to invokeTxn(T)). The events of different transactions may be interleaved in any order in a derived committed history. A committed history S is *sequential* if it contains no overlapping transactions, namely, events of different transactions do not interleave each other. S is *legal* if each read of a data item returns the last value written to that data item (or its initial value if no write operation was applied to it so far). An execution of a PDTS is *serializable* if it derives some legal sequential committed history. A PDTS is serializable if all its executions are serializable.

Weak Progress. A transactional system must guarantee at least *weak progress*: every transaction is eventually *decided*, and every transaction that did not execute concurrently with any other transaction eventually *commits*.

5.2.1 Multicore Scalability Properties

To scale to many processes on each server node, transactional systems should reduce memory contention between different transactions. This topic has been extensively studied in the literature on parallel transactional systems [AHM11; AH12; AF15; Boy14; CKZ+13; GK08; IR94; PPR+15]. Here, we focus on two well-known properties, disjoint-access parallelism and invisible reads, that are known to reduce contention and improve scalability in parallel systems. We later show how they interact with distributed scalability properties.

Disjoint-Access Parallelism

Originally introduced to describe the degree of parallelism of implementations of shared memory primitives [IR94], and later adapted to transactional memory, *disjoint-access parallelism* intuitively means that transactions that are disjoint at a high level, e.g., whose data sets do not intersect, do not contend on shared memory accesses [AHM11; PPR+15]. While this property may sound intuitive, it can in fact be difficult to achieve, as it forbids the use of global locks

or other global synchronization mechanisms. Multiple versions of disjoint-access parallelism exist in the literature, differing in which transactions are considered to be disjoint at a high level. Here, we use the following definition.

Definition 5.2.1 (disjoint-access parallelism (DAP))**.** An implementation of a PDTS satisfies *DAP* if two transactions whose data sets do not intersect cannot contend.

Invisible Reads

The second property we consider, *invisible reads*, intuitively requires that transactions' read operations not execute any shared memory writes. This property greatly benefits workloads with read hotspots, by dramatically reducing cache coherence traffic. Two variants of this property are common in the literature. The first, which we call *weak invisible reads*, only requires invisible reads at the granularity of transactions. That is, if a transaction is read-only (i.e., its write set is empty), then it may not make any changes to the shared memory. This simple property has been often used in the literature [AHM11; PPR+15].

Definition 5.2.2 (Weak invisible reads)**.** An implementation of a PDTS satisfies *weak invisible reads* if, in all its executions, every transaction with an empty write set does not execute any non-trivial primitives.

However, this property is quite weak, as it says nothing about the number of shared memory writes a transaction may execute once it has even a single item in its write set. When developing systems that decrease coherence traffic, this is often not enough. Indeed, papers that refer to invisible reads in the systems literature [SWL+20; TZK+13a] require that no read operation in the transaction be the cause of shared memory modifications. Note that an algorithm that locally stores the read set for validation (which is the case in the above referenced systems) can still satisfy invisible reads, since the writes are not to shared memory. Attiya et al. [AH12] formalize this stronger notion of invisible reads by requiring that we be able take an execution E and replace any transaction T in E with a transaction that has the same write set but an empty read set, and arrive at an execution that is indistinguishable from E. Intuitively, this captures the requirement that reads should not update shared metadata (e.g., through "read locks"). We adopt Attiya et al.'s definition of invisible reads here, adapted to fit our model.

Definition 5.2.3 (Invisible reads (adapted from [AH12]))**.** An implementation I of a PDTS satisfies the *invisible reads* property if it satisfies weak invisible reads and, additionally, for any execution E of I that includes a transaction T with write set W and read set R, there exists an execution E' of I identical to E except that it has no steps of T and it includes steps of a transaction T', which has the same interval as T (i.e., T's first and last steps in E are replaced by T''s ones in E'), and writes the same values to W in the same order as in T, but has an empty read set.

Note that the invisible reads property complements the DAP property for enhanced multicore scalability. A system that has both allows all transactions that do not conflict, not just

the disjoint-access ones, to proceed independently, with no contention (as we will show in Lemma 5.4.2). Interestingly, previous works discovered some inherent tradeoffs of such systems [AHM11; PPR+15], in conjunction with strong progress guarantees. In this work, we study these properties under a very weak notion of progress, but with added requirements on distributed scalability (see Section 5.3).

5.3 Multinode Performance Properties

To overcome the limitations of a single machine (e.g., limited resources, lack of fault tolerance), distributed transactional systems shard or replicate the data items on multiple nodes, and, thus, must incorporate distributed algorithms that coordinate among multiple nodes. The performance of these distributed algorithms largely depends on the number of communication rounds required to execute a transaction. Ideally, at least in the absence of conflicts, transactions can be executed in few rounds of communication, even if some nodes experience failures. In this section we propose formal definitions for a few multinode performance properties.

5.3.1 Distributed Disjoint-Access Parallelism

We start by proposing an extension of DAP to distributed algorithms, which we term *distributed-DAP*, or DDAP. In addition to requiring DAP, DDAP proscribes transactions from contending on a node unless they access common elements that reside at that node:

Definition 5.3.1 (Distributed disjoint-access parallelism (DDAP))**.** An implementation of a PDTS satisfies *distributed disjoint-access parallelism (DDAP)* if for any two transactions T and T', and any node N_i, if T and T''s data sets do not intersect on node N_i (i.e., $D_T \cap D_{T'} \cap \Sigma_i = \emptyset$), then they do not contend on node N_i.

While the main goal of sharding is to distribute the workload across nodes, DDAP links sharding to increased parallelism – DDAP systems can offer more node parallelism than DAP systems through sharding.

5.3.2 Fast Decision

Distributed transactional systems must integrate agreement protocols (such as *atomic commitment* and *consensus*) to ensure consistency across all nodes involved in transaction processing. Fast variants of such protocols can reach agreement in two message delays in "good" executions [Lam06c]. Ideally, we would like distributed transactional systems to preserve this best-case lower bound, and decide transactions in two message delays; reducing the number of message delays required to process transactions not only can significantly reduce the latency as perceived by the application (processing delay within a machine is usually smaller than the delay on the network), but can also reduce the *contention footprint* [FTA14] of the

transactions (intuitively, this is the duration of time in which a transaction might interfere with other transactions in the system).

Requiring transactions to be decided in just two message delays, however, is too restrictive in many scenarios. The latency of a distributed transactional system depends on how many message delays are required for a transaction to "learn" its data set (data items and their values); the data set needs to be returned to the application when the transaction commits, and is also used to determine whether the transaction can commit. For example, for interactive transactions or disaggregated storage, the values must be made available to the application (which runs in a client process) before the transaction can continue to execute. Thus, since the data items are remote, each read operation results in two message delays, one to request the data from the remote node and one for the remote node to reply. For non-interactive transactions or systems where transaction execution can be offloaded to the node processes, the latency for learning the data set can be improved; since the client does not need to immediately know the return value of read operations, the values of data items can be learned through a chain of messages that continue transaction processing at the nodes containing the remote data. More precisely, the client first determines a node, n_1, that contains the first data item the transaction needs to read; the client sends a message to n_1 containing the transaction; n_1 processes the transaction, preforming the read locally, until it determines that the transaction needs to perform a remote read from another node, n_2; n_1 sends a message to n_2 containing the transaction and its state so far; n_2 continues processing the transaction, performing the read locally, and so on. RPC chains [SAKM09] already provides an implementation of this mechanism, saving one message delay per remote read operation. At the lowest extreme, non-sharded transactional systems can learn a transaction's entire data set in a single message delay.

We introduce the *fast decision* property to describe distributed transactional systems that can decide each transaction in "good" executions within only two message delays in addition to the message delays it requires to "learn" the transaction's entire data set. As explained above, the number of message delays required to learn a transaction's data set depends on several design choices. We note that often, deciding a transaction's outcome within two message delays after learning its data set is not plausible if the execution has suboptimal conditions, for example, if there are transactional conflicts that need to be resolved, or if not all nodes reply to messages within some timeout. This is true even for just consensus, where the two-message-delay decisions can happen only in favorable executions, on a *fast path* [ABG+20; Lam06a]. We therefore define the fast decision property to only be required in such favorable executions.

To formalize fast decisions, we must be able to discuss several intuitive concepts more formally. In particular, we begin by defining the *depth of a transaction*, to allow us to formally discuss the number of message delays that the transactional system requires to decide a transaction.

Definition 5.3.2 (Depth of a transaction)**.** The *depth of a decided transaction* T in execution

E of a PDTS implementation, $d_E(T)$, is the depth of the response step of T's coordinator handler in E.

In many cases, we need to refer to the depth of a transaction T in an execution in which T is still ongoing, and its coordinator handler has not reached its response step yet. While we could simply refer to the depth of the deepest step of T in the execution, this would not be appropriate: it is possible that a transaction in fact took steps along one 'causal path' that led to a large depth, but when the response step to T's coordinator handler happens, its depth is actually shorter. In such a case, we really only care about the depth along the 'causal paths' that lead to the response step, since these are the ones affecting the latency to the application. To capture this notion, we define the *partial depth* of a transaction T in a prefix of an execution in which T is decided as follows.

Definition 5.3.3 (Partial depth of a transaction)**.** Let T be a decided transaction in execution E of a PDTS implementation. The *partial depth* of T in a prefix P of E in which T is not decided, $d_E(T, P)$, is the maximum step depth across all steps associated with T in P, which happened-before the response step of T's coordinator handler in E (or 0 if there are no such steps).

We next formalize another useful concept that we need for the discussion of fast decisions; namely, what it means to *learn* the data set of a transaction. For that purpose, we introduce the following two definitions:

Definition 5.3.4 (Decided data item)**.** A data item d is *decided* to be in a transaction T's read or write set in execution E of a PDTS implementation if, in all extensions of E, the read or write set respectively in the return value of invokeTxn(T) contains d.

Definition 5.3.5 (Decided value)**.** A data item d's *value* is *decided* for T in execution E of a PDTS implementation if, in all extensions of E, the read set in the return value of invokeTxn(T) contains d and with the same value.

Note that a data item's value can be decided for a transaction only if that data item is part of its read set; the definition does not apply for data items in the write set. In the definition of the fast decision property and in the proofs, we refer in most places to knowing the decided values and not the data items in the write set as well. This is because knowing the read set and its values implies that a transaction's write set is decided in case it commits; this is the property that matters in many of the arguments we use in this work.

Finally, we are ready to discuss the *fast decision* property. Intuitively, the formal definition of the property considers *favorable* executions, which are synchronous, failure-free and have each transaction run solo. For those executions, the property requires two things to hold: first, a transaction is not allowed to spend more than two message delays without learning some new value for its data set, and second, once its entire data set is known, it must be decided within 2 more message delays (Corollary 5.1). This captures 'speed' in both learning the data

set and deciding the transaction outcome. As discussed above, 2 message delays is an upper bound on the minimal amount of time needed to perform a read operation (and bring its value to the necessary process). Note that this is a tight bound for systems processing interactive transactions, and as such, fast decision also means optimal latency for these systems.

Definition 5.3.6 (Fast decision)**.** A PDTS implementation I is *fast deciding* if, for every failure-free synchronous execution E of I and every decided transaction T in E that did not execute concurrently with any other transaction, for any prefix P of E such that $d_E(T, P) < d_E(T) - 2$, there exists a prefix of E of partial depth $d_E(T, P) + 2$ in which the number of values known by some process to be decided is bigger than in P.

Formalizing the allowed depth of a transaction in terms of prefixes of an execution in which the transaction is already decided (so we know its depth in that execution) helps capture the two requirements we want: (1) for any prefix of the execution, if we advance from it by two message delays, we must have improved our knowledge of the values of the read set, and (2) once the read set and its values are completely known (regardless of the depth of the prefix in which this occurs), we must be at most 2 message delays from deciding that transaction. Corollary 5.1 helps make this intuition concrete.

Corollary 5.1. *For every failure-free synchronous execution E of a fast-deciding PDTS implementation I and every decided transaction T in E that did not execute concurrently with any other transaction, let P be the shortest prefix of E in which the value of each item in T's read set is known by some process to be decided. Then*

$$d_E(T) \leq d_E(T, P) + 2.$$

(Intuitively, T must be decided within at most 2 message delays from when T's read set including its values are known to be decided.)

Proof Assume by contradiction that $d_E(T) > d_E(T, P) + 2$. Then by the fast decision property of I, there exists a prefix of E in which the number of data items whose value is known by some process to be decided is bigger than in P. But this is impossible, since the values of T's entire read set are known to be decided in P. ■

Several fast-deciding distributed transactional systems have been recently proposed for general interactive transactions [KPF+13; ZSS+15; SWL+20]; our fast decision property captures what they informally refer to as "one round-trip commitment". These systems use an *optimistic concurrency control* and start with an execution phase that constructs their data sets with two message delays per read operation. The agreement phase consists of validation checks that require a single round-trip latency (integrates atomic commitment and a fast consensus path in one single round trip). The write phase happens asynchronously, after the response of the transaction has been emitted to the application.

5.3.3 Seamless Fault Tolerance

High availability is critical for transactional storage systems, as many of their applications expect their data to be always accessible. In other words, the system must mask server failures and network slowdowns. To achieve this, many systems in practice are designed to be fault tolerant; the system can continue to operate despite the failures of some of its nodes.

However, oftentimes, while the system can continue to function when failures occur, it experiences periods of unavailability, or its performance degrades by multiple orders of magnitude while recovering [ABG+20; WJC+17]. This is the case in systems that must manually reconfigure upon failures [VS04], and those that rely on a leader [ABG+20; OO14; WJC+17; Lam01; Lam98].

These slow failure-recovery mechanisms, while providing some form of guaranteed availability, may not be sufficient for systems in which high availability is truly critical; suffering from long periods of severe slowdowns potentially from a single server failure may not be acceptable in some applications.

To address this issue, some works in recent years have focused on designing algorithms that experience minimal slowdowns, or no slowdowns at all, upon failures. One approach has been to minimize the impact of leader failures by making the leader-change mechanism light-weight and switching leaders even when failures do not occur [YMR+19]. Another approach aims to eliminate the leader completely; such algorithms are called *leaderless* algorithms [ADG+21; SWL+20; ZSS+15; MAK12]. All of these approaches aim to tolerate the failure of some nodes without impacting the latency of ongoing transactions.

In this work, we formalize this goal of tolerating failures without impacting latency into a property that we call *s-seamless fault tolerance*, where $s \leq f$. In essence, s-seamless fault tolerance requires that if only up to s failures occur in an execution, no slowdown is experienced. To capture this formally, we require that for any execution E with up to $s-1$ crashes, it be possible to find an equivalent execution E' with one more crash event, which may happen at any time after the crashes in E, where the depth of all transactions are the same in E and E'. We express this in an inductive definition.

Definition 5.3.7 (s-seamless fault tolerance)**.** Any implementation of a PDTS satisfies 0-seamless fault tolerance. An implementation I of a PDTS satisfies *s-seamless fault tolerance* if it satisfies $(s-1)$-seamless fault tolerance, and for any execution E of I with $s-1$ node crashes, for any prefix E_P of E that contains the $s-1$ node crashes, and any node crash event c of a node that has not crashed in E_P, there exists an execution E' of I whose prefix is $E_P \cdot c$, such that (1) stripping each of E and E' of all steps other than invocation and response steps of coordinator handlers results in the same sequence of invocation and response steps (intuitively, the executions are equivalent), and (2) the depth of each decided transaction is the same in both executions (intuitively, E' seamlessly tolerates the node crashes).

While s-seamless fault tolerance offers the extremely desirable robustness property, it also requires that: a) no single node can be on the critical path of all transactions, and b) no single

node can be solely responsible for processing a transactional task. This can be a double-edged sword; on the one hand, this eliminates the possibility of a leader bottleneck, which implies better scalability. On the other hand, it disallows certain optimizations, like reading from a single replica.

5.4 Impossibility Results

Having specified some key properties which make distributed transactional systems fast and scalable, we now turn to the main result of our work: unfortunately, there is a tension between these multinode performance properties and the single-node multicore performance properties discussed in Section 5.2. More specifically, we present the **FIDS** theorems, which formalize the impossibility of achieving all of these properties simultaneously in two different parallel distributed settings.

5.4.1 The FIDS Theorems

The first **FIDS** theorem states that no PDTS with weak progress which *shards data* can guarantee **F**ast decision, **I**nvisible reads, distributed **D**isjoint-access parallelism, and **S**erializability simultaneously. This is in contrast to known systems that achieve just the multinode properties [SWL+20; ZSS+15; MNLL16] or just the multicore properties [YPSD16; TZK+13a; DNN+15]. Thus, the FIDS theorem truly shows tensions that arise when a transactional system is both parallel and distributed. This version of the FIDS theorem considers only systems that shard data, that is, systems in which each node only stores part of the database items. Interestingly, the impossibility holds in this setting even without requiring any fault tolerance, and in particular, without seamless fault tolerance. We note that the FIDS theorem applies also to systems that replicate data in addition to sharding it; adding replication on top of a sharded system only makes it more complex. Formally:

Theorem 5.2 (The FIDS theorem for sharded transactional systems). *There is no implementation of a PDTS which shards data across multiple nodes that guarantees weak progress, and simultaneously provides fast decision, invisible reads, distributed disjoint-access parallelism, and serializability.*

For systems that maintain multiple copies of the data, but do not necessarily shard it, we show a different version of the result. Note that in such systems, distribution comes from replication; several nodes, each with a copy of the entire database, are used to ensure fault tolerance. For this setting, we present the **R-FIDS** theorem: a PDTS with weak progress that utilizes client-driven replication and satisfies **R**obustness to at least one failure through the seamless fault tolerance property, in addition to satisfying **F**ast decision, **I**nvisible reads, **D**isjoint-access parallelism, and **S**erializability, is also impossible to implement. Formally:

Theorem 5.3 (The R-FIDS theorem for replicated transactional systems). *There is no implementation of a PDTS that utilizes client-driven replication that guarantees weak progress, and*

simultaneously provides 1*-seamless fault tolerance, fast decision, invisible reads, disjoint-access parallelism, and serializability.*

Next we present an overview of the proof technique for the two versions of the FIDS theorem, followed by a detailed proof for each of them and the supporting lemmas.

5.4.2 Proof Overview

Both proofs have a similar structure; we consider example transactions that form a dependency cycle, and show an execution in which all of them commit, thereby violating serializability. To argue that all transactions in our execution commit, we build the execution by merging executions in which each transaction ran solo (and therefore had to commit by weak progress), and showing that the resulting concurrent execution is indistinguishable to each transaction from its solo run. Starting with solo executions also gives us another property that we can exploit; we define the solo executions to be synchronous and failure-free, and therefore they must be fast deciding as well.

The key challenge in the proofs is how to construct a concurrent execution E_{concur} that remains indistinguishable to all processes from the solo execution that they were a part of. To do so, we divide the concurrent execution into two phases; first, we let the solo executions run, in any interleaving, until right before the point in each execution at which some process learns the values of its transaction's read set. When this point is reached in each solo execution, we carefully interleave the remaining steps in a second phase of the concurrent execution. A key feature is that by the fast decision property, which each solo execution satisfies, once some process learns the read set including its values, there are at most two message delays left in each solo execution before the transaction is decided. This bounds the amount of communication we need to worry about in the second phase of the concurrent execution.

To show that E_{concur} is indistinguishable from the solo runs, we look at each of the two phases separately. The idea is to show that no process makes *any* shared memory modifications in the first phase, and then show that we can interleave messages and message handlers in a way that allows each transaction to be oblivious to the other transactions for at least one more intuitive 'round trip', which is all we need to reach decision according to the fast decision property.

To show that a transaction performs no shared memory modifications in the first phase of the concurrent execution we construct, we rely on the way we choose the transactions, their data sets, and when in the execution their data sets are decided; in both proofs, the transactions we choose may have empty or non-empty write sets, depending on the results of their reads. The following lemma shows that as long as a transaction's write set is not *known* to be non-empty, the transaction cannot cause any modifications in a system that provides weak invisible reads.

Lemma 5.4.1. *Let I be an implementation of a PDTS that provides weak invisible reads, and let T be a transaction in an execution E of I, such that no process in E knows the following*

proposition: T's write set is non-empty in all extensions of this execution in which T is decided. Then T cannot cause any base object modifications in E.

This lemma, combined with the way we choose the transactions in our proofs, immediately implies that phase 1 of E_{concur} is indistinguishable to all processes from the solo executions they are a part of.

The proofs differ somewhat in how they show that E_{concur} is indistinguishable from the solo runs in the second phase. We argue about restricted shared memory modifications through the use of the DAP and invisible reads properties in the following key lemma, which intuitively shows that transactions that do not conflict do not (visibly) contend.

Lemma 5.4.2. *Let I be an implementation of a PDTS that provides both DAP and invisible reads, and let T be a transaction in an execution E of I, such that its final write set is W. Then T does not cause any base object modifications visible to any concurrent transaction in E whose data set does not overlap with W.*

To make phase 2 of E_{concur} also indistinguishable from the solo executions, we schedule the remaining messages carefully. In particular, we schedule messages sent by reading transactions to each node before those sent by writing transactions, and again rely on DAP and invisible reads to argue that the reading transactions' handlers will not cause changes visible to those who write afterwards. However, here the two proofs diverge.

Sharded Systems

We first discuss the proof structure for showing that serializable sharded transactional systems that provide weak progress cannot simultaneously achieve fast decision, invisible reads and distributed disjoint-access parallelism (DDAP). That is, sharding the data across multiple nodes while achieving these properties is impossible even if we do not tolerate any failures (Theorem 5.2).

The proof uses two nodes and two transactions, each reading from a data item on one node and, if it sees the initial value, writing on the other node. The read set of one transaction is the same as the (potential) write set of the other transaction. We need to argue that the reading transaction on some node cannot cause modifications on that node that are visible to the writing transaction. However, since the write set of each transaction overlaps with the data set of the other, we cannot apply Lemma 5.4.2. Instead, we rely on DDAP, and show that with this property, the reading transaction indeed cannot be visible to the writing one on each node. We show a lemma very similar to Lemma 5.4.2 but which applies to transactions whose write set on a specific node does not overlap the data set of another transaction on that node.

Lemma 5.4.3. *In any implementation of a PDTS that provides both DDAP and invisible reads, a transaction whose write set is W does not cause any modifications on shared based objects on a node N visible to any concurrent transaction whose data set does not overlap with W on N.*

The proof of this lemma is very similar to the proof of Lemma 5.4.2. The only required adjustments are using *DDAP* instead of DAP, and referring to T''s data set and T's write set and modification *on a certain node* N.

Note that while the proof of the FIDS theorem relies on sharding, it does not need fault tolerance. In particular, it does not make use of the seamless fault tolerance property. However, the result does apply to systems in which the data is both sharded and replicated, as those systems are even more complex than ones in which no replication is used.

Replicated but Unsharded Systems

So far, we have considered a PDTS in which node failures cannot be tolerated; if one of the nodes crashes, we lose all data items stored on that node, and cannot execute any transactions that access those data items. However, in reality, server failures are common, and therefore many practical systems use replication to avoid system failures. Of course, the impossibility result of Theorem 5.2 holds for a PDTS even for the more difficult case in which failures are possible and each node's data is replicated on several backups.

However, we now turn our attention to PDTSs in which the entire database is stored on each node. This setting makes it plausible that a client could get away with accessing only one node to see the state of the data items of its transaction. However, we show that the impossibility of Theorem 5.2 still holds in this setting for a system in which failures are tolerated without affecting transaction latency (i.e., systems that satisfy seamless fault tolerance) (Theorem 5.3).

As explained in Section 5.4.2, the use of seamless fault tolerance requires us to explicitly argue about the length of the executions in which transactions decide. To do so, we need the following lemma, which gives a lower bound for the depth at which a transaction's read set and values can be decided.

Lemma 5.4.4. *There is no execution E of any serializable PDTS implementation that tolerates at least 1 failure in which there is a transaction T and prefix P such that $d_E(T, P) < 2$ and some process knows the decided value of some read of T in P.*

Once we have this lemma, the proof of the R-FIDS theorem is then similar to the proof of the FIDS theorem. We build a cycle of dependencies between transactions where each neighboring pair in the cycle overlaps on a single data item that one of them reads and the other writes. The key is that because of invisible reads, each read can happen before the write on the same data item without leaving a trace. However, to construct this cycle in the replicated case, we need at least 3 replicas, 3 transactions and 3 data items. This is because we can no longer separate the read and write of a single transaction on each node. Furthermore, we make use of Lemma 5.4.4, as well as the budgeted depth of a transaction in a fast-deciding execution, to explicitly argue about the amount of communication possible after a transaction learns its write set.

More specifically, we choose three transactions, where the write set of one equals the read set of the next. We divide them into pairs, where within each pair, the write set of one does

not overlap with the data set of the other. We can then directly use Lemma 5.4.2 to argue that the second one to be scheduled of this pair will not see changes made by the first. We exploit fault tolerance to have the third transaction's messages never reach that node. However, here, we must be careful, since we defined the solo executions to be failure-free to guarantee fast decisions. We therefore rely on seamless fault tolerance; we show indistinguishability of the concurrent execution not from the original solo executions, but from executions of the same depth that we know exist due to seamless fault tolerance.

Interestingly, when we convert a solo execution S to an execution F of the same depth (but with a node failure) via the seamless fault tolerance property, we may lose its fast decision property. That is, while the new execution must have the same depth as the original ones, that does not guarantee that it will also be fast deciding, as the fast decision property does not solely refer to the length of the execution. In particular, it could be the case that in F, the data set of a transaction including its values is learned earlier, but then the transaction takes more than 2 message delays to be decided. This would be problematic for our proof, in which the indistinguishability relies heavily on fast decision once the data set including its values are known. To show that this cannot happen in the executions we consider, we rely on Lemma 5.4.4 that bounds the depth at which any transaction in a fault tolerant system can learn the decided values of its reads.

5.4.3 Full Proofs

Lemma 5.4.1. *Let I be an implementation of a PDTS that provides weak invisible reads, and let T be a transaction in an execution E of I, such that no process in E knows the following proposition: T's write set is non-empty in all extensions of this execution in which T is decided. Then T cannot cause any base object modifications in E.*

Proof Let I be an implementation of a PDTS that satisfies weak invisible reads. Assume by contradiction that there is a transaction T in execution E of I, such that no process in E knows that T's final write set is not empty in all extensions in which T is decided, and a process p runs a handler associated with T that performs some base object modification.

Since p does not know that T's final write set is not empty in all extensions of the current execution in which T is decided, there exists an execution indistinguishable to p from E that has an extension in which T's final write set is empty. Let that extension be $E_{readOnly}$. Since T's final write set in $E_{readOnly}$ is empty, then by weak invisible reads, T cannot cause base object modifications in $E_{readOnly}$. Contradiction. ■

Lemma 5.4.2. *Let I be an implementation of a PDTS that provides both DAP and invisible reads, and let T be a transaction in an execution E of I, such that its final write set is W. Then T does not cause any base object modifications visible to any concurrent transaction in E whose data set does not overlap with W.*

Proof Let I be an implementation of a PDTS that satisfies DAP and invisible reads. Let T be a transaction whose final write set in an execution E of I is W. Assume by contradiction

that there exists some transaction T' concurrent with T in E whose data set does not overlap with W, but which sees a modification made by T in E. That is, there is some base object operation step s of T' whose return value is affected by T's modification.

By invisible reads, there exists an execution E' of I identical to E except that it includes a transaction T_{noRead} in place of T with the same interval, where T_{noRead} has W as its write set and an empty read set. By DAP, T_{noRead} does not modify in E' any base object accessed by any concurrent transaction whose data set does not overlap with W. In particular, T_{noRead} cannot make any modifications visible to T' in E'. Note that step s must exist in E', since by definition, E' is identical to E except in steps associated with T and T_{noRead}. However, in E, s's return value is affected by T's modification, and in E', this modification does not exist. Therefore, E' cannot be an execution of I. Contradiction. ■

Lemma 5.4.3. *In any implementation of a PDTS that provides both DDAP and invisible reads, a transaction whose write set is W does not cause any modifications on shared based objects on a node N visible to any concurrent transaction whose data set does not overlap with W on N.*

Proof Let I be an implementation of a PDTS that satisfies DDAP and invisible reads. Let T be a transaction whose final write set on a node N in an execution E of I is W. Assume by contradiction that there exists some transaction T' concurrent with T in E whose data set on N does not overlap with W, but which sees a modification made by T on a base object on N in E. That is, there is some base-object operation step s of T' whose return value is affected by T's modification.

By invisible reads, there exists an execution E' of I identical to E except that it includes a transaction T_{noRead} in place of T with the same interval, where T_{noRead} has W as its write set on N and an empty read set. By DDAP, T_{noRead} does not modify in E' any base object on N accessed by any concurrent transaction whose data set does not overlap with W on N. In particular, T_{noRead} cannot make any modifications visible to T' on N in E'. Note that step s must exist in E', since by definition, E' is identical to E except in steps associated with T and T_{noRead}. However, in E, s's return value is affected by T's modification, and in E', this modification does not exist. Therefore, E' cannot be an execution of I. Contradiction. ■

Lemma 5.4.4. *There is no execution E of any serializable PDTS implementation that tolerates at least 1 failure in which there is a transaction T and prefix P such that $d_E(T, P) < 2$ and some process knows the decided value of some read of T in P.*

Proof Assume by contradiction that there is some implementation I of a serializable PDTS that tolerates at least 1 failure, an execution E of I, and a prefix P of E such that $d_E(T, P) \leq 1$ and some process knows the decided value of some data item d of T in P. Without loss of generality, let process p on node N be the process that knows d's decided value, let that value be v and let T's invoking client be C. Note that since C does not have access to the data, and any step of any process not on C's node must be of depth at least 1, p cannot be on C's node, and cannot have received any message from any process other than C within depth less than

2. Therefore p can only know the value of d on node N, but not any other nodes. Consider the following executions.

E_{N-fail}. E_{N-fail} and E are identical up to right before T's invocation. In E_{N-fail}, node N fails at this point. Then, a transaction T' is invoked by a client $C' \neq C$. T' writes a value $v' \neq v$ to d and commits. After T' commits, T is invoked in E_{N-fail}. Clearly, by serializability, T's read of d in E_{N-fail} returns v' or a more updated value, but not v.

E_{N-slow}. E_{N-slow} is identical to E_{N-fail} except that node N does not fail in E_{N-slow}. Instead, all messages to and from N are arbitrarily delayed in E_{N-slow} starting at the same point at which N fails in E_{N-fail}. Clearly, E_{N-slow} is indistinguishable from E_{N-fail} to all processes not on N.

E'. E' is identical to E_{N-slow} except that node N receives messages from client C. Clearly, E' and E_{N-slow} are indistinguishable to all processes not on N. So, T's read of d must return the same value as in E_{N-slow}, namely v' or a more updated one, but not v. However, note that E' is also indistinguishable to processes on N from E in any prefix of E of partial depth < 2 for T, since no process in N received any messages other than those it received in E, and since clients do not receive any messages not related to their own transactions, so C must have sent the same message(s) to N in E' as it did in E. Therefore, there is a prefix P' of E' indistinguishable to p from P, in which v is not the decided value of d, contradicting p's knowledge of d's decided value in P. ■

Theorem 5.2 (The FIDS theorem for sharded transactional systems). *There is no implementation of a PDTS which shards data across multiple nodes that guarantees weak progress, and simultaneously provides fast decision, invisible reads, distributed disjoint-access parallelism, and serializability.*

Proof Assume by contradiction that there exists an implementation I of a PDTS with all the properties in the theorem statement. Consider a database with 2 data items, X_1, X_2, partitioned on 2 nodes, N_1, N_2 respectively. Consider two transactions, T_1, T_2, with the following data sets: T_1's read set is $\{X_1\}$. Its write set is $\{X_2\}$ if its read returns the initial value of X_1, in which case it writes a value different from X_2's initial value. Otherwise, its write set is empty. For T_2, its read set is $\{X_2\}$, and its write set is $\{X_1\}$ if its read returns the initial value of X_2, and empty otherwise. If its write set is non-empty, it writes a value different from X_1's initial value. Let T_1 be executed by a client C_1 and T_2 be executed by a different client C_2. Consider the following executions.

Solo Executions. We define two executions S_1, S_2, corresponding to T_1, T_2 respectively running in isolation, without the other transaction present in the execution. Both executions are synchronous and failure-free. By weak progress, T_i commits in S_i, and by serializability, T_i returns the initial value of its read item and therefore its write set is not empty.

Concurrent Execution. We define an execution, E_{concur}, where T_1 and T_2 execute concurrently. On each node, each transaction is executed on different processes. Recall that this can happen since this is a parallel system, and the executing processes for a transaction are arbitrarily chosen among the idle processes of each node. In E_{concur}, for each transaction

T_i, we let each process that executes it run until right before it knows the decided read set and read set value of T_i. Let the prefix of E_{concur} that includes all these steps be P_1. We then let each process that handles T_i run until when the next step of its handler has depth $\geq d_{S_i}(T_i) - 2$. Next, we let all messages sent on behalf of T_1 to N_1 and not yet received reach N_1 and be handled before any message sent on behalf of T_2 to N_1. For node N_2, we let the reverse happen; messages sent on behalf of T_2 reach it and are handled before messages sent on behalf of T_1. Finally, we resume all processes, and pause node processes that handle T_i when the next step of their handler has depth $\geq d_{S_i}(T_i)$. As for the client of each transaction, we let any messages sent to it arrive in the same order as they did in their corresponding solo executions (we will show that it receives the same messages in E_{concur}).

We now claim that execution E_{concur} is indistinguishable to C_i from S_i, and indistinguishable to each node process running T_i from the prefix of S_i containing all this process's steps of depth $< d_{S_i}(T_i)$. To do so, we consider the execution in two phases; the phase before the two transactions achieve knowledge of their data sets including their values (up to the end of P_1), and the phase afterwards.

Phase 1 of E_{concur}. Note that for any prefix P of E_{concur} in which T_i's read set's value is not known to be decided by some process, T_i's known decided write set in P is empty. Consider the longest prefix $P_{undecided_i}$ of P_1 in which the decided write set of T_i is still empty. Note that for every process p, its knowledge of T_i's write set in $P_{undecided_i}$ is the same as it is in P. Therefore, by Lemma 5.4.1, in any such prefix P, T_i may not make any modifications to shared base objects visible to *any* concurrent transaction. Therefore, in phase 1 there are no modifications visible to either transaction that were not visible in the solo execution as well. Thus, by the end of phase 1, E_{concur}'s prefix P_1 is indistinguishable to both transactions from their respective solo executions. Therefore, both transactions read the initial values of their respective read sets, and both have a non-empty write set in E_{concur}.

Phase 2 of E_{concur}. To show that E_{concur} remains indistinguishable from the solo executions to their respective transactions in phase 2, we rely on the order of messages that are received by the two nodes.

First, we note that by Lemma 5.4.3, T_i does not make base object modifications visible to $T_{(i \bmod 2)+1}$ on node N_i, since T_i's final write set is $\{X_{(i \bmod 2)+1}\}$, which does not intersect $T_{(i \bmod 2)+1}$'s final data set on node N_i.

Next, note that in each solo execution S_i, the first process that knows the decided value of T_i must be on node N_i, since that is where the data for the read of T_i is stored. Furthermore, by construction of E_{concur}, any messages sent on behalf of T_i to N_i immediately after both transactions gain knowledge of their write sets arrives before any such message sent on behalf of $T_{(i \bmod 2)+1}$, and its handler is completely executed. Thus, by the above claim, on both nodes, all handlers of both transactions for messages sent at depth $d_{E_{concur}}(T_i, P_1)$ execute to completion in a way that is indistinguishable to T_i from the solo execution S_i.

Finally, note that since S_i is synchronous and failure and conflict free, and I satisfies the fast decision property, by Corollary 5.1, the depth of T_i in S_i is at most 2 more than the partial depth of the first prefix in which T_i's data set including its values became known.

Transactions read and write sets			
T	T_1	T_2	T_3
R_T	$\{X_2\}$	$\{X_3\}$	$\{X_1\}$
W_T	$\{X_1\}$ if R(X_2)=$\perp$, else {}	$\{X_2\}$ if R(X_3)=$\perp$, else {}	$\{X_3\}$ if R(X_1)=$\perp$, else {}

In particular, since S_i is indistinguishable to processes executing T_i from E_{concur} up to that point, this means that $d_{S_i}(T_i) \leq d_{E_{concur}}(T_i, P_1) + 2$. Thus, once messages from within the handlers that were activated by messages sent in E_{concur} at depth $d_{E_{concur}}(T_i, P_1)$ are received, T_i must be decided in E_{concur} as well, since E_{concur} is indistinguishable from S_i to all processes running T_i up to this point. Therefore, both transactions commit successfully in E_{concur} in a manner indistinguishable from their respective solo executions.

However, this yields a circular dependency between the two transactions; T_2 must occur before T_1, since it returns the initial value of X_2, before T_1 writes to it. Similarly, T_1 must occur before T_2, since it returns the initial value of X_1. This therefore contradicts serializability. ■

Theorem 5.3 (The R-FIDS theorem for replicated transactional systems). *There is no implementation of a PDTS that utilizes client-driven replication that guarantees weak progress, and simultaneously provides* 1*-seamless fault tolerance, fast decision, invisible reads, disjoint-access parallelism, and serializability.*

Proof Assume by contradiction that there exists an implementation I of a parallel replicated transactional system with all the properties stated in the theorem.

Consider a transactional system with 3 nodes N_1, N_2, N_3. (For less than 3 nodes, there is no PDTS that tolerates $f \geq 1$ failures in the partial-synchrony model [DLS88].) Further consider 3 transactions T_1, T_2, T_3, 3 client processes C_1, C_2, C_3, and 3 data items X_1, X_2, X_3 each of which is replicated on all 3 nodes. The data sets of the transactions are as follows: T_i's read set includes $X_{(i \bmod 3)+1}$, and if the result of T_i's read of $X_{(i \bmod 3)+1}$ is the initial value of $X_{(i \bmod 3)+1}$, its write set includes X_i. Otherwise, its write set is empty. Each transaction T_i, if its write set is non-empty, writes a value that is different from X_i's initial value.

Consider the following executions. For each $i = 1, 2, 3$, in any of the following executions, if it includes T_i then its coordinator handler is executed by C_i.

Solo Executions. Let E_1, E_2, E_3 be failure-free synchronous executions of I, where transaction T_i runs solo in E_i. Since E_i contains a single transaction and I satisfies weak progress, transaction T_i commits in E_i. Since E_i is synchronous, has no failures and contains only T_i, and I satisfies fast decision, T_i is fast deciding in E_i.

Claim: T_i must have a depth of at most 4 in E_i.

To see this, note that by the definition of fast decision, if transaction T_i in E_i has depth at least 3, the empty prefix of E_i must have an extension C_i of partial depth $d_{E_i}(T_i, C_i) \leq 2$ in

which the value of the read set's item is known by some process to be decided, and therefore the write set is known by that process to be decided as well. By Corollary 5.1, the depth of T_i in E_i must be at most 2 more than the depth of C_i, and therefore is at most 4.

Since I satisfies 1-seamless fault tolerance, there exist executions E'_1, E'_2, E'_3 of I, where the first event in E'_i is a crash of N_i, T_i runs solo and the depth of T_i in E'_i is the same as its depth in E_i. We assume that in each E'_i, a different set of processes runs the handlers. Lastly, since I is serializable, each E'_i is serializable, thus T_i's read in E'_i returns the initial value of $X_{(i \bmod 3)+1}$, and therefore modifies X_i as part of its write set.

Since T_i's write set is only determined from the outcome of T_i's read, and may be empty until that read's value is decided, by Lemma 5.4.1, no base object modifications visible to other transactions are executed by T_i in E'_i until after T_i's read set values are known to some process. Let the shortest prefix at which some process gains knowledge of T_i's read set values in E'_i be P_i. By Lemma 5.4.4, $d_{E'_i}(T_i, P_i) \geq 2$.

Concurrent Execution. We define an execution E_{concur} with all 3 transactions. In E_{concur}, all messages between processes on node N_i and any process that executes handlers associated with T_i are arbitrarily delayed. For each $i = 1, 2, 3$, let processes that execute T_i in E'_i run in E_{concur} identically to E'_i, in the same order of steps, until the end of P_i.

Note that up to this point, E_{concur} is indistinguishable to all executing processes from the solo executions, since none of them has made any shared memory modifications visible to the others. Therefore, the prefix $P_{knowledge}$ of E_{concur} up to this point is an execution of I.

We continue E_{concur} as follows: Let all messages sent on behalf of T_i at depth $d_{E'_i}(T_i, P_i)$ be sent in E_{concur}, and be received and handled in the following order: on node N_1, messages for T_2 are received first, and their handlers are run to completion, followed by messages for T_3. On node N_2, T_3's messages are handled first, followed by messages of T_1. Finally, on node N_3, messages of T_1 are handled first followed by messages of T_2. coordinator handlers receive messages in the same order they received them in their corresponding solo executions.

Recall that transaction T_i reads data item $X_{(i \bmod 3)+1}$ and, if it reads the initial value, writes data item X_i. Thus, the service order defined above for execution E_{concur} (see the order in which the nodes process their writes in Figure 5.2) means that on each node, the second serviced transaction writes to data item X after the first transaction reads X, but it is never the case that a transaction reads a data item after it was written by another transaction on the same node. Since the data set of the second transaction to execute handlers after prefix $P_{knowledge}$ on each node does not overlap with the write set of the first one, and since I

	X_1	X_2	X_3
N_1		1	2
N_2	2		1
N_3	1	2	

Figure 5.2: Visual representation of execution E_{concur} in the proof of Theorem 5.3. The numbers in the table represent the order of writing on each node; on node N_1, X_2 is written first, followed by X_3, and so on.

provides invisible reads and DAP, then by Lemma 5.4.2, the first transaction does not make base object modifications visible to the second transaction. In other words, on each node, a process executing the second transaction cannot observe any changes on shared memory. Thus, E_{concur} is still indistinguishable from E'_i to any node process that executes T_i up to the end of the handlers of messages sent at depth $d_{E_{concur}}(T_i, P_i)$. Note, however, that since $d_{E'_i}(T_i) \leq 4$ and $d_{E_{concur}}(T_i, P_i) \geq 2$, this means that E_{concur} remains indistinguishable from E'_i to these processes until T_i is decided.

Therefore, for all three transactions T_i commit in E_{concur}, reading the initial value of $X_{(i \bmod 3)+1}$ and writing a non-initial value in X_i. However, this yields a circular dependency between the transactions (transaction T_1 must happen before T_2, which must happen before T_3, which must happen before T_1), which contradicts serializability. ■

5.5 Possibility Results

In this section, we show that any subset of the properties outlined in Theorem 5.3 *is* possible to achieve simultaneously in a single system. To do so, we present four distributed transactional system algorithms, each sacrificing one of the desired properties. Recall from our model description that the presented protocols work under the assumption that the client does not fail and nodes do not recover, and as such are not intended to be used "as is" in practice.

Our algorithms all have a similar structure, with minor tweaks to guarantee or sacrifice certain properties. We therefore begin by presenting a 'base' algorithm. This algorithm achieves all the desired properties (i.e., fast decision, invisible reads, DDAP, and seamless fault tolerance), but is not serializable. In each of the following subsections, we tweak the base algorithm to sacrifice one desirable property and gain serializability. We note that we refer here to DDAP though Theorem 5.3 refers to DAP, since DDAP is stronger than DAP (it implies DAP), and so our possibility results are accordingly stronger using DDAP.

The base algorithm works as follows. Each data item d is replicated on $0 < k <= n$ nodes, to which we refer as d's *replica group*. If $k < n$ then we assume the system is sharded. All nodes execute the same state machine both in the sharded and the fully replicated case (in which $k = n$). Each data item maintains its current value and a sequence number. Each data item is also assigned two locks: a short-lived lock, $lock_S$, that ensures that the value and the sequence number of the data item are read and written atomically, and a long-lived lock, $lock_L$, utilized by the distributed concurrency control mechanism. We adopt an interactive transaction model for executing transactions, where the initiating client coordinates its own transaction. There are two phases per transaction:

Execution phase. During the execution phase the client reads items and dynamically constructs its data set. A process receiving an execution phase message from a client waits until the relevant data item $lock_S$ is unlocked, and then reads the data item and sends its value and sequence number to the client. To ensure that the sequence number and the data item's value are read atomically while still maintaining the invisible reads property (e.g., there

are no read locks), we employ a lock-free read mechanism, as follows. To read a data item, the reader checks that its $lock_S$ is not acquired, then checks the sequence number, and then verifies that the lock is still not taken. If that verification passes, it now reads the data item's value, and finally checks the sequence number one more time. If the sequence numbers are the same and the lock was free on both checks, it succeeds. Otherwise, it fails (the reader can retry or send an abort message to the client). Writers first acquire the lock on the data item, then change its sequence number, and only then modify the value. This mechanism is similar to the one used in Silo [TZK+13a], and guarantees atomic reads of the sequence number and value, even if they cannot be read atomically together in hardware. During this execution phase, the client records the sequence number of each data item it reads.

Validation phase. The client then executes a validation phase in which it communicates with the nodes to verify that its read values are still valid (the data items have the same sequence number) and to update the items in its write set. A process receiving a validation phase message on node N_i iterates through the data items d in the message such that d is stored in node N_i; for each item in the read set, the process checks whether the data item d's long-lived lock $lock_L$ is locked, and whether its current sequence number matches the one specified by the client. For each item in the write set, the process tries to acquire d's $lock_L$. If at any point, it runs into a data item that is already locked or its sequence number is out of date, it releases the data items it has already locked (if any), and replies 'abort' to the client. Otherwise, if all data items' sequence numbers matched the client's validation phase message and (if it's not a read-only transaction) the process managed to lock them itself, the process

```
110 //Execution phase
111 for each key r in the read set:
112     Send ⟨READ,r⟩ to all nodes that have r
113     Wait to receive (val, seqNum) from k-f nodes
114     Record for r the (val, seqNum) of the message with the max seqNum

116 Create transaction message T=⟨tid,sequence⟩, where sequence specifies the data item accesses,
        ↪ with reads of the form (key, seqNum) and writes of the form (key, newVal)

118 //Validation Phase
119 Send ⟨VALIDATE,T⟩ to all nodes
120 Wait for n-f responses
121 if all responses are of COMMIT type:
122     for each key w in the write set:
123         Append seqNum to w in T, where seqNum = max seqNum for w across all responses +
              ↪ 1
124     Commit T
125     Send ⟨COMMIT,T⟩ to all nodes
126 else:
127     Abort T
128     Send ⟨ABORT,T⟩ to all nodes
```

Figure 5.3: Client code in the base algorithm.

sends 'commit' to the client.

The client waits to receive $n - f$ replies from each shard, where f is the number of failures the algorithm tolerates (at most $(k-1)/2$ for k replicas of each shard). If any of them was 'abort', the transaction is aborted. Otherwise, the transaction is committed. The client then lets all nodes know whether the transaction committed by sending another message. Note that this message does not affect the fast decision property, since this is after the client knows the outcome of the transaction. A process receiving this final message applies the writes to the transaction's write set if it committed, and then releases all locks. We present pseudocode for this algorithm split by client code (Figure 5.3) and node process code (Figure 5.4).

It is easy to see that this algorithm satisfies all 4 desired properties; each data item has its own associated base objects, and a transaction only accesses data item d if d is in its data set. Therefore, DDAP holds. Furthermore, non-trivial primitives are only applied to base objects associated with a data item in a transaction's write set. In particular, only the write set is locked, and the value and sequence numbers are only updated on locked data items. Thus, the invisible reads property is satisfied; any two transactions whose write sets are the same execute non-trivial primitives on the same set of base objects. The other two properties hold because of the way the client operates; the client only sends one message and waits for one response from $n - f$ nodes in the validation phase before it knows the outcome of the transaction. Furthermore, the execution phase causes only trivial primitives on shared memory, and therefore sends no non-trivial messages. Thus, the fast decision and f-seamless fault tolerance properties also hold. However, the impossibility results apply to this algorithm, since it is not serializable (it provides the read committed isolation guarantee; the lack of serializability can be checked by running the scenarios presented in our impossibility proofs).

5.5.1 Sacrificing Fast Decision

In this subsection we show how to achieve serializability by sacrificing the fast decision property. To do so, we split the validation phase of the client into two round trips (4 message delays), similarly to [DNN+15], but leave the rest of the algorithm the same. In this version, the client first sends a message with just its write set, and the nodes attempt to lock all data items in that set. If all nodes are successful, the client then sends its read set for the nodes to verify that the sequence numbers have not changed. If all nodes observe the same sequence number for all data items in the read set, then the client can commit. Otherwise, the transaction is aborted. Clearly, this algorithm does not fast decide, but does not damage the other properties of the base algorithm we considered.

Proof of serializability. We now briefly argue that the algorithm satisfies serializability. Recall that the client first reads all items in its read set and learns their sequence number. It then acquires all write locks in the first round of the validation phase, and then rechecks the sequence numbers of read items in the second round. Note that data items change if and only if their sequence number changes as well. Thus, if no sequence number changed in the read set, then none of these items have been modified by any other process between the time

```
129 Upon receiving ⟨READ,r⟩ from C:
130     Let d be the data item with key r
131     // lock-free atomic read of r's seqNum and val
132     wait until d.lock_S is not locked
133     int seqNum = d.seqNum
134     if d.lock_S is locked: retry
135     data val = d.val
136     if seqNum != d.seqNum: retry
137     Send (val, seqNum) to C

139 Upon receiving ⟨VALIDATE,T⟩ from C:
140     bool success = true
141     List⟨key,seqNum⟩ writeSeqs = []
142     for each data item d corresponding to an item of T:
143         if d.lock_L != None:
144             success = false; break
145         if d is in the read set of T:
146             if d.seqNum != seqNum of d in T:
147                 success = false; break
148         else:
149             if not d.lock_L.CAS(None, T.tid):
150                 success = false; break
151             Append ⟨d.key,d.seqNum⟩ to writeSeqs
152     if success:
153         Send ⟨COMMIT,writeSeqs⟩ to C
154     else:
155         Set all successfully-CASed lock_L to None
156         Send ABORT to C

158 Upon receiving ⟨COMMIT,T⟩ from C:
159     for ⟨k,newVal,seqNum⟩ in the write set of T:
160         Let d be the data item with key k
161         d.lock_S.lock()
162         if d.seqNum < seqNum:
163             d.seqNum = seqNum
164             d.val = newVal
165         d.lock_S.unlock()
166         if d.lock_L == T.tid:
167             d.lock_L = None

169 Upon receiving ⟨ABORT,T⟩ from C:
170     for each data item d corresponding to a write set item of T:
171         if d.lock_L == T.tid:
172             d.lock_L = None
```

Figure 5.4: Process code in the base algorithm.

the client read the last item in the execution phase and the time it read the first one in the validation phase. Thus, the transaction can serialize at the point at which it holds all write

locks.

For the replicated case, as long as $f < k/2$, any two client quorums will intersect at at least one node of each replica group of each data item common to both transactions; this guarantees that concurrent conflicting transactions cannot both commit.

5.5.2 Sacrificing Invisible Reads

We now present a simple transactional memory algorithm that satisfies serializability, DDAP, fast decision, f-seamless fault tolerance, and *weak* invisible reads. That is, we show that weakening the invisible reads property suffices to make such a system feasible.

To relax the invisible reads property, but guarantee serializability, we tweak the base algorithm to acquire more locks. In particular, instead of just acquiring the locks of data items in a transaction's write set, processes now also acquire the locks for the data items in the read set. However, to achieve the weak invisible reads property, this is only done for transactions that have a non-empty write set. The rest of the algorithm behaves exactly like the base algorithm. This algorithm is very similar to the one presented by Szekeres et al. in Meerkat [SWL+20]. The main difference here is that we avoid non-trivial primitives by read-only transactions, to maintain the weak invisible reads property.

Clearly, this algorithm preserves DDAP, fast decision, and f-seamless fault tolerance. Furthermore, by construction, it is clear that it satisfies the weak invisible reads property.

Proof of Serializability. To argue about serializability, we show a reduction to strict two-phase locking (S2PL) [BHG86], which is known to produce serializable executions. S2PL acquires read and write locks on all data items in the transaction's data sets and releases them only after commit. We note that this is the behavior of transactions in our algorithm if they are not read-only. Thus, any non-read-only transaction in our algorithm can serialize. We now only have to argue about read-only transactions. Note that this was already covered in the argument for the algorithm in Section 5.5.1; since the client reads items in its read set before the validation phase, and then checks that all sequence numbers remain unchanged during the validation phase, these data items cannot have been modified during the time between the client's last read in the execution phase and its first read in the validation phase. Thus, read-only transactions can serialize during that time.

5.5.3 Sacrificing Seamless Fault Tolerance

We now present an algorithm that satisfies all properties outlined in this work except seamless fault tolerance. Note that this property only makes sense for the replicated case; unreplicated distributed systems do not tolerate any failures. Also, note that the algorithm only provides DAP, as opposed to DDAP, since we proved that achieving DDAP with the other properties is impossible when the system is sharded (Theorem 5.2).

Furthermore, note that seamless fault tolerance is closely related to fast decision. In particular, an algorithm that is fast deciding but does not satisfy seamless fault tolerance may

stop satisfying the fast decision bounds not only if there are transaction conflicts, but also if there are node failures.

We capitalize on this fact in our solution. To create a serializable transactional system that satisfies all desirable properties except for even just 1-seamless fault tolerance, we employ the base algorithm almost as-is; the only change is that this time, the client waits for *all* nodes to reply, rather than just $n - f$. This solution is serializable, since each node handles all transactions, so conflicting transactions will compete for the same lock.

To still allow fault tolerance (though not seamless), we allow the client to time out on the nodes. If the client times out while waiting to hear replies from all nodes, then the algorithm defaults to the one presented in Section 5.5.1. That is, the client sends a signal to all nodes letting them know it is switching gears, and restarts executing its current transaction, but this time waiting for just $n - f$ node responses, and using a two-round validation phase. When a process receives such a restart signal from the client, it releases all locks it held for the current transaction. The correctness of this algorithm follows directly from the correctness of the algorithm in Section 5.5.1.

5.5.4 Sacrificing Distributed Disjoint-Access Parallelism

Guaranteeing the rest of the properties without DDAP is simple; we use the base algorithm, but maintain a global lock per node, rather than a lock per data item. Transactions always lock all nodes, even if they do not have any data item on some of them. The only exception is read-only transactions, which do not acquire locks, but check that the relevant lock is free and that the data items on the node have the same sequence number as they did in the execution phase. Clearly, this solution maintains fast decision and seamless fault tolerance, but does not preserve DDAP. Note that since read-only transactions do not acquire locks, the weak invisible reads property is preserved. Furthermore, since all writing transactions acquire the locks of all nodes, transactions with the same write set execute non-trivial primitives on the same set of objects: all of the locks, and the sequence numbers and values of each data item in their write set. Thus, the invisible reads property is preserved as well.

The serializability argument for this algorithm is very similar to that of the algorithm in Section 5.5.2. Non-read-only transactions acquire the locks on their entire data set, and commit only after successfully doing so. Thus, they are serializable by S2PL [BHG86]. Read-only transactions can serialize sometime between the last read in the execution phase and the first in the validation phase.

5.6 Related Work

Disjoint-access parallelism was first introduced in [IR94] in the context of shared memory objects. It was later adapted to the context of transactions. Over the years, it has been extensively studied as a desirable property for scalable multicore systems [YPSD16; TZK+13b; SWL+20; AHM11; AF15; PPR+15; GK08]. Several versions of DAP have been considered, dif-

fering in what is considered a conflict between operations (or transactions). A common variant of DAP considers two transactions to conflict if they are connected in the conflict graph of the execution (where vertices are transactions and there is an edge between two transactions if their data sets intersect) [AHM11; PPR+15]. In this work, we consider a stricter version, which only defines transactions as conflicting if they are neighbors in the conflict graph. This version has also appeared frequently in the literature [PPR+15; BDFG14].

Invisible reads have also been extensively considered in the literature [SS05; AHM11; SWL+20; TZK+13b; YPSD16; HLMS03]. Many papers consider invisible reads on the granularity of data item accesses; any read operation on a data item should not cause changes to shared memory [TZK+13b; SWL+20; HLMS03]. Others, often those that study invisible reads from a more theoretical lens, consider only the invisibility of *read-only transactions* [AHM11; PPR+15].

Some impossibility tradeoffs for transactional systems, similar to the one we show in this work, are known in the literature. Attiya et al. [AHM11] show that it is impossible to achieve weak invisible reads, disjoint-access parallelism, and wait-freedom in a parallel transactional system. Peluso et al. [PPR+15] show an impossibility of a similar setting, with disjoint-access parallelism, weak invisible reads, and wait-freedom, but consider any correctness criterion that provides real-time ordering. Bushkov et al. [BDFG14] show that it is impossible to achieve disjoint-access parallelism and obstruction-freedom, even when aiming for consistency that is weaker than serializability. In this work, none of our algorithms provide the obstruction-freedom considered in [BDFG14]; we use locks, and our algorithms can therefore indefinitely prevent progress if process failures can occur while holding locks.

Fast paths for *fast decision* have been considered extensively in the replication and consensus literature [KR01; DGL05; Lam06a; ABG+19; ABG+20]. In most of these works, the conditions for remaining on the fast path include experiencing no failures. That is, they do not provide seamless fault tolerance. However, some algorithms, like Fast Paxos [Lam06a], can handle some failures without leaving the fast path. In the context of transactional systems, the fast path is often considered for conflict-free executions rather than those without failures [SWL+20; ZSS+15; MNLL16; KPF+13], as we do in this work. Seamless fault tolerance captures the idea that (few) failures should not cause an execution to leave the fast path. Systems often have a general fault tolerance f that is higher than the number of failures they can tolerate in a seamless manner [SWL+20; ZSS+15; MNLL16; KPF+13].

Seamless fault tolerance as presented here is also related to *leaderlessness* [MAK12; ADG+21], as any leader-based algorithm would slow down upon a leader failure. However, the leaderless requirement alone is less strict than our seamless fault tolerance; Antoniadis et al. [ADG+21] defined a leaderless algorithm as any algorithm that can terminate despite an adaptive adversary that can choose which process to temporarily remove from an execution at any point in time. This does not put any requirement on the speed at which the execution must terminate.

Parallel distributed transactional systems have been recently studied in the systems literature. Meerkat [SWL+20] provides serializability and weak progress, and three of the desirable properties we outline in this work. It does not, however, provide invisible reads in any form

(not even the weaker version). Eve [KWQ+12] considers replication for multicore systems. It briefly outlines how PDTS transactions are possible using its replication system, but it is not their main focus.

5.7 Discussion

This work is inspired by recent trends in network capabilities, which motivate the study of distributed transactional systems that also take advantage of the parallelism available on each of their servers. We formalize three performance properties of distributed transactional systems that have appeared intuitively in various papers in the literature, and show that these properties have inherent tensions with multicore scalability properties. In particular, in this work we formalized the notions of distributed disjoint-access parallelism, a fast decision path for transactions, and robustness in the form of seamless fault tolerance. Combined with the well-known multicore scalability properties of disjoint-access parallelism and invisible reads, we present the *FIDS theorem*, and its fault tolerant version, the *R-FIDS theorem*, which show that serializable transactional systems cannot satisfy all these properties at once. Finally, we show that removing any one of these properties allows for feasible implementations.

We note that our possibility results can be seen as "proofs of concept" rather than practical implementations. It would be interesting to design practical algorithms that give up just one of the properties we discuss. We believe that each property has its own merit for certain applications and workloads, and it would be interesting to determine which property would be the best to abandon for which types of applications.

In this work, we focused on studying parallel distributed transactional systems under a minimal progress guarantee. It would also be interesting to explore PDTSs under stronger progress conditions, or consistency conditions other than serializability. It would be equally interesting to see if the tension still exists between weaker variants of the properties we considered.

Chapter 6

Durable Queues: The Second Amendment

This chapter is based on the work presented at [SP21a] and [SP21b] (and the code is available at https://github.com/galysela/DurableQueues).

6.1 Introduction

Byte-addressable non-volatile memory combines the byte-accessibility of DRAM with the durability and size of storage. Various technologies, such as resistive random access memory [AS10], phase-change memory [RBB+08] and 3D XPoint [Int19], are expected to become available soon, with Intel/Micron 3D XPoint already available to consumers (under the brand name Optane). NVRAM is expected to co-exist or replace DRAM in upcoming architectures, allowing program's modifications to its data structures survive system crashes. NVRAM platforms are expected to make a fundamental change in the design of the computing infrastructure including file systems, databases and other computations that process persistent data.

While data stored in main memory will survive a crash, without further technological development, the caches and machine registers remain volatile, losing their content during a crash. This creates a consistency challenge, because writes may not reach the memory at the time and order the processor issues them. When programs write data to memory, the CPU does not access the memory directly, but rather writes to the cache and the data only later gets flushed back to memory. Furthermore, the order in which cache lines get written back to the memory is unpredictable, as cache line evictions are triggered by local needs to make room for new cache content. This process may cause the state of the memory after a crash to become inconsistent, reflecting some modifications but missing others, impeding correct recovery.

In order to make sure that the memory contains the required data for a potential crash and recovery, special instructions are used to force the flushing of cache lines from the cache to the memory. Asynchronous flush instructions initiate a cache line flush and let other instructions proceed while the data is being copied to memory. An additional synchronous fence (such

as Intel's Store Fence (SFENCE) instruction) makes sure that the flushing becomes visible before any other memory instruction becomes visible to other threads. The fence instruction is blocking and costly and therefore durable algorithms have attempted to reduce the use of SFENCE to achieve better performance. Cohen et al. [CGZ18] have shown that a durably linearizable [IMS16] lock-free [Her91] object must use at least one fence instruction per update operation at worst case. They also presented a universal construction that achieves this bound, but their universal construction was intended as a proof of existence and no attempt was made to provide acceptable performance.

The initial goal of this project was to optimize the performance of a durable FIFO queue. FIFO queues are used at the core of several existing persistent messaging systems (e.g., IBM MQ [IBM], Oracle Tuxedo MQ [Ora], Rabbit MQ [Sof] and many more). Currently these queues are structured to suit the block-based interface of HDDs and SSDs. This design incurs costs like marshaling queue updates in streams, file system calls to persist message queues, etc., and so an adaptation to NVRAM platforms can bring a dramatic improvement to the queues performance and future use.

Following previous work in this area, we focused on reducing the number of blocking persist operations. We started with the lock-free Michael-Scott queue (MSQ) [MS96], which was used in previous work [FHMP18] due to its wide applicability to all architectures. We amended MSQ in two different manners, obtaining two novel durably linearizable lock-free queue constructions with a minimal number of blocking persist operations: one blocking persist operation for any data structure modification operation. This meets the lower bound of Cohen et al. [CGZ18]. These two optimal durable queue algorithms are the first contribution of this work.

One of these two algorithms, called UnlinkedQ, is designed in the spirit of [ZFS+19] to avoid persisting the underlying node links. In this algorithm, we allocate the nodes on designated areas, in which the recovery procedure can look for valid nodes of the queue. This requires a new persistent ordering mechanism that allows the recovery to determine the order of nodes in the queue without incurring a large overhead on the normal execution of the queue. The second algorithm, denoted LinkedQ, does persist the underlying node links. It reduces the number of fences by using a validity scheme to inform the recovery algorithm which nodes are adequate for recovery. It also adds a backward link to the queue nodes, for enabling to efficiently assist persisting concurrent operations.

We implemented these two algorithms on a platform with an Intel Cascade Lake processor and an Intel Optane NVRAM. Surprisingly, the new algorithms did not show a clear improvement over the state-of-the-art durable queue of Friedman et al. [FHMP18] although Friedman's queue executes more blocking persist operations during the execution. Further investigations raised an interesting problem. Our queues frequently access flushed cache lines, and these accesses significantly deteriorated performance. It turned out that Intel flush instructions, which flush a cache line to the NVRAM, cause the flushed cache line to be invalidated in the cache, so that subsequent accesses yield cache misses and re-read the data from memory. (We tried various instructions including the most advanced Cache Line Write

Back (CLWB) instruction, but they all had the same performance degradation effect). The resulting additional loads from memory are significantly more costly on NVRAM than on DRAM, due to the high NVRAM read latency. While the recently-launched Intel Ice Lake processors with Optane persistent memory 200 series may provide flush instructions that do not invalidate the flushed cache lines, existing NVRAM architectures with Cascade Lake processors do not seem to support such instructions. Our impression is corroborated in the findings of [WCD+20; KAK20; vRVL+19; Hao19; BDY+21; FBW+20; FPR21]. Existing (costly) architectures will probably remain in use for years to come and one needs to use algorithmic modifications to obtain improved performance on such machines.

We amended the two algorithms further, obtaining algorithms that avoid accessing flushed locations. While changing the algorithms, we made sure that their original advantage of a single fence per update operation is maintained. An evaluation of this second amendment demonstrates a significant performance improvement, which confirms the high cost of accessing flushed content on these platforms.

The second contribution of this work is a guideline for designing durable data structures and algorithms for NVRAM. In addition to the well-known guideline to minimize blocking persist operations, we recommend designing algorithms with reduced access to recently flushed cache lines[1]. This guideline is relevant for platforms that invalidate cache lines when flushing their content to the memory, and the purpose is to avoid the cost of fetching data from the memory after it is evicted from the cache. This guideline is especially important in light of the high read latency of available NVRAM (see measurements in [vRVL+19; YKH+20]).

Our third contribution is the design of durable lock-free queues with significantly improved performance for the new Intel Optane architecture. We present OptUnlinkedQ and OptLinkedQ, obtained by amending UnlinkedQ and LinkedQ respectively according to the new guideline. OptUnlinkedQ and OptLinkedQ are the best performing lock-free durable queues available today. We compare the performance of OptUnlinkedQ and OptLinkedQ against state-of-the-art durable queues and against UnlinkedQ and LinkedQ themselves, which use minimal blocking persist operations but do not consider the new guideline and do not reduce access to flushed cache lines. While OptUnlinkedQ and OptLinkedQ outperform all other queues on nearly all thread counts and workloads, we believe UnlinkedQ and LinkedQ are still interesting to present. This is because for potential more advanced platforms that might provide flushing without cache invalidation, UnlinkedQ and LinkedQ may turn out best.

From a theoretical standpoint, it is interesting to note that OptUnlinkedQ and OptLinkedQ yield the best possible design characteristics for durability. Following our guideline above, they make zero accesses to content that was previously (explicitly) flushed, while they also meet the lower bound shown by Cohen et al. [CGZ18], executing only a single blocking persist operation per data structure update operation. Interestingly, while these theoretical

[1]We consider only explicitly flushed cache lines. There are additional implicit flushes, e.g., when the system evacuates cache lines to make space for new lines that need to be loaded to the cache. Such implicit flushes are hard to predict and this guideline does not attempt to consider them.

characteristics are the best possible, they are also obtainable for any object. This follows from the universal construction of [CGZ18]. While Cohen's universal construction of lock-free durably linearizable data structures is not practical, it has the above-mentioned characteristics (a single blocking persist instruction per update operation and no access to flushed content) and it is applicable to any object.

The rest of the chapter is organized as follows. In Section 6.2 we elaborate on the model and the general upper bound on the design parameters. In Section 6.3 we recall the definitions of durable linearizability and lock-freedom as well as MSQ, the basic queue algorithm that we extend in our constructions. We discuss related work in Section 6.4. In Section 6.5 we provide describe the main ideas in the first amendment to MSQ: minimizing blocking persist operations, which produces UnlinkedQ and LinkedQ. In Section 6.6 we describe the second amendment to the two algorithms, adhering to the guideline of reducing access to flushed data, which results in the optimal queues OptUnlinkedQ and OptLinkedQ. We argue about the durable linearizability and lock-freedom of our queues in Sections 6.7 and 6.8. The memory management scheme applied in our queues is described in Section 6.9, and the performance of all queue algorithms is evaluated in Section 6.10. We conclude in Section 6.11.

6.2 Model

In the persistent memory model, there are two levels of memory – volatile (registers, caches) and persistent (NVRAM). Values in the cache may be written back to the persistent memory implicitly by a cache eviction, or explicitly by flush instructions. We adopt the failure model of Izraelevitz et al. [IMS16] for crashes, which considers full-system crashes in which all processes fail together. The state of the volatile memory is lost in a crash, but the state of the persistent memory remains unaffected. After a crash, new threads are created and proceed with the computation. Each data structure may provide a recovery procedure to be invoked after the crash for restoring a consistent state of the object from its preserved state in the NVRAM. Our data structures apply a complete recovery before continuing with any new operation.

To maintain correctness in the presence of crashes, one has to ensure that necessary writes propagate from the cache to the persistent memory. To ensure a written value becomes persistent (after being written to the cache), one may issue a flush instruction and block until it completes. A flush instruction receives a memory address and flushes the content of the cache line containing this address to the persistent memory. Some flush instructions are asynchronous, enabling issuing multiple flushes concurrently as an optimization. Subsequently, a store fence instruction, denoted SFENCE (like the instruction name on Intel), may be placed to ensure completion of all previous asynchronous flushes. Throughout the thesis, when mentioning a *persisting* of a location, we refer to an asynchronous flush of its address accompanied by an SFENCE to ensure that the data in this location has been written to the NVRAM.

Intel flush instructions (such as the synchronous Flush Cache Line (CLFLUSH) and the asynchronous Flush Cache Line Optimized (CLFLUSHOPT) and CLWB) take a mem-

ory location and write back the cache line containing it to the memory, if this line consists of modified data. According to the Intel architectures software developer's manual [Int20], CLFLUSH and CLFLUSHOPT do not only write the cache line to the memory, but rather also invalidate it. Regarding CLWB, the Intel manual states that it *may* retain the line in the cache. However, on the Second Generation Intel Xeon Scalable Cascade Lake processor we use, CLWB seems to invalidate the cache line like CLFLUSHOPT does: replacing CLWB with CLFLUSHOPT in all the data structures we measured yielded similar performance. This is also noted by others [e.g., WCD+20; KAK20; vRVL+19; Hao19; BDY+21; FBW+20; FPR21]. The recently-launched Third Generation Intel Xeon Scalable Ice Lake processors with Optane persistent memory 200 series may implement CLWB retaining lines in the cache, but NVRAM platforms currently in the market do not seem to support flushes without cache invalidation. Existing architectures will probably remain in use for years to come. Therefore, designers of efficient durable algorithms should take into consideration the cost of accessing a memory location after it was flushed and evicted from the cache.

To eliminate some of the costly persisting occurrences, we rely on the following assumption, which is based on the cache line granularity of write backs to memory. The assumption is mentioned in the SNIA NVM programming model [SNI17, Section 10.1.1], adopted by Intel for working with persistent memory (as stated in Intel's formal persistent memory programming book [Sca20]), and is confirmed by Intel Senior Principal Engineer Andy Rudoff in online informal discussions [e.g., Rud19; Rud20; BR21]. This assumption was also previously made in [CBB14, Footnote 16] and [CFL17, Assumption 2].

Assumption 6.2.1. A cache line is evicted atomically to memory, thus, the order of multiple writes to the same cache line is preserved in memory. In other words, the content of a cache line in the memory reflects a prefix of the stores to that cache line.

As the order of writes to the same cache line is preserved in NVRAM[2], placing a flush plus SFENCE between them to ensure their persistence order (which is required for writes to different cache lines) is redundant.

In addition to a flush, another useful instruction for our algorithms is an instruction that writes back data directly to the memory without touching or polluting the cache (like Store Doubleword Using Non-Temporal Hint (MOVNTI)). Such asynchronous instructions require an accompanying SFENCE to ensure their completion.

6.2.1 Upper Bound on Accesses after a Flush

Due to cache invalidation after a flush, we recommend designing algorithms that minimize accesses to flushed content. This comes in addition to designing algorithms that minimize

[2]Writes to the cache are not guaranteed to occur in program order, due to compiler optimizations, but program order can be enforced by placing inexpensive release fences (that prevent compiler optimizations, thus, ordering writes to cache). We placed release fences in our implementation where required, and we do not further mention them here.

blocking flushes. In fact, we claim that it is possible to implement any object with a deterministic sequential specification in a durably linearizable lock-free way using the minimum possible number of fences (one per update operation and zero per read-only operation, as proved by [CGZ18]) while at the same time performing zero accesses to (explicitly) flushed cache lines.

To prove our claim, we leverage the universal construction of [CGZ18], called Order Now, Linearize Later (ONLL). ONLL consists of two main components. The first is a shared execution trace, containing a mark indicating the trace's prefix guaranteed to be persistent. This prefix represents the current state of the object. The execution trace is not used during recovery, thus also not persisted to memory. The second component is local per-thread persistent logs (adopted from [CFL17]), that will be read during recovery. An update operation first appends a record representing it to the execution trace, then appends a copy of the trace's suffix that is not yet guaranteed to be persistent to its local log and persists it, and finally marks the trace's prefix up to the current operation as persistent. A read-only operation calculates its response based on the current state of the object, represented by the trace's marked prefix.

[CGZ18] proves that ONLL obtains the minimum possible number of fences. We suggest the following slight modification to ONLL: align log entries to cache lines, so that no two entries will share a cache line. By applying this modification, ONLL still obtains minimum fences, while also performing no access to flushed memory. This is because only data in the local per-thread persistent logs is explicitly flushed, and these logs' cache lines are not accessed after their flush: they are read only during recovery, and not written by following log appends – which write to following cache lines thanks to our modification.

6.3 Preliminaries for the Durable Queues

6.3.1 MS-Queue

Our persistent queue algorithms extend the widely used MSQ (the Michael and Scott queue [MS96]), a well-performing concurrent queue adequate for general hardware, included as part of the Java™ Concurrency Package [Lea09]. This is a (non-persistent) lock-free FIFO queue, which supports enqueue and dequeue operations. It implements the queue as a singly-linked list with head and tail pointers. Nodes in the list have two fields: a value and a next pointer. The head points to the first node of the list, which functions as a dummy node. Subsequent nodes, after the dummy and until the node whose $next$ pointer's value is NULL, contain the queue's items. The queue is initialized to an empty queue as a list that contains a single (dummy) node, to which both the head and tail point.

A dequeue operation checks if $next$ of the obtained head is NULL (meaning the queue is empty). If so, this is a *failing dequeue* that returns without extracting an item from the queue. Otherwise, an attempt is made to update the head to point to its successive node in the list, using a CAS, and on failure the dequeue operation starts over. A dequeue that succeeds to perform a CAS that advances the queue's head is denoted a *successful dequeue*.

Enqueuing requires two CAS operations. Initially, a node with the item to enqueue is created. Then, an attempt to set $tail$->$next$ to the address of the new node is made using a first CAS. The CAS fails if the value of $tail$->$next$ is not NULL in that moment. In such a case, an attempt to advance $tail$ to the current value of $tail$->$next$ is made using a CAS, to help an obstructing enqueue operation complete. Then, a new attempt to perform the first CAS starts. After the first CAS succeeds, a second CAS is applied to update $tail$ to point to the new node.

6.3.2 Linearizability and Durable Linearizability

Defining correctness for durable executions in the presence of both concurrency and NVRAM is not a trivial task. In this work, following recent work in this domain, we adopt durable linearizability [IMS16] described below as a correctness criterion. Nevertheless, it is easy to verify that our proposed queues satisfy also other correctness criteria, like strict linearizability [AF03], persistent atomicity [GL04] and recoverable linearizability [BGT15].

In Chapter 2 we brought the linearizability definition and some relevant basic terminology. We now bring the adjustments required to define correctness in a system that ensures durability. In the full-system-crash model, an execution of a concurrent system may be modeled by a finite sequence of events of three types: invocation events and response events, each tied to specific process and object, and system crash events (which are not tied to a specific process or object). Such sequence is denoted a *history*. A history in the full-system-crash model (i.e., a history that might contain crashes in which all processes fail together and there is no subsequent thread reuse) is considered *durably linearizable* [IMS16] if the history with the crashes omitted is linearizable.

6.3.3 Lock-Freedom

We brought the definition of lock-freedom [Her91] in Chapter 2. We now extend the definition to executions with crashes, and define a concurrent object implementation to be lock-free in the presence of crashes if each time a thread executes an operation on the object, and there are no interrupting crash events since the operation's invocation, some thread (not necessarily the same one) completes an operation on the object within a finite number of steps. This definition is equivalent to the one brought in [ZFS+19], which considers crashes as progress, as if a crash is one of the operations on the data structure. Lock-freedom guarantees system-wide progress. Our implementations are lock-free.

6.4 Related Work

There has been a large body of work by multiple communities that provides algorithms for NVRAM. Several libraries for persistent transactional access to objects in NVRAM have been proposed [CCA+11; CFR18; KPS+16; MMT+18; RCFC19; tea; VTS11; WRL19; ZZLS19], but persistent transactions require heavy-duty logging mechanisms, and thus do not yield highly

efficient solutions, and are not competitive with ad-hoc constructions such as ours. [MIS20] presents an NVRAM library taking another logging-based approach. Izraelevitz et al. [IMS16] suggested to automatically make concurrent objects durably linearizable by adding a flush and a fence after each access to global memory (a read or a write). This transformation yields a durable variant of any existing lock-free data structure, but the resulting implementations are typically inefficient. The first ad-hoc efficient lock-free durable data structure was the queue presented by Friedman et al. [FHMP18], with a substantial reduction of the number of fences executed with each operation over the general construction of Izraelevitz. Subsequently, David et al. [DDGZ18] presented lock-free durable set implementations (including a linked list, a skip list and a hash map). Zuriel et al. [ZFS+19] improved over that construction and presented a set with a single SFENCE per update operation, thus meeting the lower bound of Cohen et al. [CGZ18] and also obtaining much better performance. Raad et al. [RWNV20] implemented a persistent FIFO queue to demonstrate the application of their suggested hardware model, but did not aim for optimized performance (e.g., they do not track the tail pointer, thus significantly slowing down enqueues).

6.5 First Amendment: Queues with Minimum Fences

The current literature offers a fast queue with several fences [FHMP18] on the one hand, and a universal construction for all data structures with a single fence (per update operation) which is extremely inefficient [CGZ18] on the other hand. A question that naturally arises is whether it is possible to reduce the number of fences to the minimum possible number, and at the same time be able to use this reduced blocking to obtain a better performing queue. In this section we provide two durably linearizable lock-free queues, UnlinkedQ and LinkedQ, that meet the theoretical lower bound on the number of fences (a single SFENCE per operation).

6.5.1 UnlinkedQ

As its name implies, UnlinkedQ does not rely on links between nodes for restoring the queue after a crash and therefore does not persist them, similarly to the basic idea in [ZFS+19]. It keeps all information required for recovery in the nodes themselves, which are located in designated areas. Upon a crash, the recovery procedure checks these nodes to decide which ones are valid and belong to the resurrected queue. The links are still used to expedite operations on the queue when no crash occurs, but they are not required to reconstruct the queue after a crash. Care is taken to persist the queue order in the nodes to allow proper recovery.

UnlinkedQ places *index* and *linked* fields in each node to enable the recovery to identify which nodes in the designated areas should be restored and in what order. The *index* field states the node's index in the queue (according to enqueue order). Overflow can be handled, but for now we allocate 64 bits for the *index* field and assume that it does not overflow (while humans are still around). The *linked* field marks nodes that have been added to the queue. After an enqueuer succeeds to link a node to the queue, it sets its *linked* flag, and then per-

sists the node content. The recovery procedure resurrects nodes that are marked *linked* and have an index larger than the head, and arranges them in the order induced by their indices. After advancing the head, a dequeuer persists the new head's index, to indicate to the recovery that all nodes up to this one are dequeued. This scheme forms a consecutive prefix of dequeued nodes – all those that the head has persistently passed, thus satisfying the FIFO order requirement.

The simple scheme described so far involves several races which should be resolved. One race stems from the fact that the order in which enqueue operations complete does not necessarily match the linking order of their nodes. For example, it is possible that the enqueue of the fourth node in the queue has completed before the enqueue of the third node in the queue completed. Hence, it might be that the fourth node is marked *linked* while the third node is not. One consequence of this race is that the indices of valid linked nodes that the recovery identifies do not always form a sequence of consecutive integers. Even worse, a dequeue operation might point the head at a node inserted by a concurrent enqueue, whose content is not yet flushed and therefore contains a stale index that may confuse the recovery.

Next, we elaborate on the implementation of UnlinkedQ, including describing how it resolves the above-mentioned issues. The UnlinkedQ algorithm is presented in Figure 6.1. A description of its operations follows.

```
173 class Node
174     Item* item
175     atomic⟨Node*⟩ next
176     bool linked
177     int index
```

```
178 Item* Dequeue()
179     while (true)
180         head = Head
181         headNext = head.ptr->next
182         if (headNext == NULL)
183             FLUSH(&Head.index); SFENCE
184             return NULL
185         if (CAS(&Head, head, ⟨headNext,
                ↪ headNext->index⟩))
186             dequeuedItem = headNext->item
187             FLUSH(&Head.index); SFENCE
188             if (nodeToRetire[tid]) // It equals
                    ↪ NULL in the first
                    ↪ successful dequeue
189                 retire(nodeToRetire[tid])
190             nodeToRetire[tid] = head.ptr
191             return dequeuedItem
```

```
192 Enqueue(item)
193     newNode = allocNode()
194     newNode->item = item
195     newNode->next = NULL
196     newNode->linked = false
197     while (true)
198         tail = Tail
199         if (tail->next == NULL)
200             newNode->index = tail->index +
                    ↪ 1
201             if (CAS(&tail->next, NULL,
                    ↪ newNode))
202                 newNode->linked = true
203                 FLUSH(newNode); SFENCE
204                 CAS(&Tail, tail, newNode)
205                 break
206         CAS(&Tail, tail, tail->next)
```

Figure 6.1: UnlinkedQ implementation

The Enqueue Operation

The enqueue operation first allocates a node and initializes its data (Lines 193–195). It then unsets *linked* (Line 196), sets the *index* of the new node to be the index of the last node plus one (Line 200), and attempts to link the node to the queue (Line 201). The reason *linked* is unset before *index* is updated, is that when the node is allocated, its *linked* flag might be set; thus, assigning the new node a relevant index in this state might erroneously cause the recovery to restore the node even though it is not yet linked to the queue.

After succeeding to link the node, the enqueuer sets its *linked* flag (Line 202), to signal to the recovery (that would run if a crash occurs) that the node should be restored. The described order of writes to the node fields guarantees, based on Assumption 6.2.1[3], that a node will be restored by the recovery only if it is successfully linked. Finally, the enqueuer persists the node and advances the queue's tail to point to the new node (Lines 203–204). If a concurrent enqueue operation interferes, the enqueuer attempts to assist the other enqueue to advance the tail to point to its node (Line 206), before starting a new attempt to enqueue its own item.

We note that the recovery procedure might restore a suffix of enqueues with nonconsecutive indices. This happens only if several enqueues are running when a crash occurs: an enqueue that linked e.g. the fourth node in the queue might have set its *linked* flag and persisted it before the crash, while an enqueue that linked the third node in the queue has not. Discarding pending enqueue operations which have not set and persisted the *linked* flag is correct due to the following observation:

Observation 6.5.1. Durable linearizability allows pending operations to not be linearized. Therefore, the recovery may discard pending enqueue operations, which might result in a suffix of enqueued nodes with nonconsecutive indices.

The Dequeue Operation

If a dequeue operation encounters an empty queue, it returns NULL. Otherwise, it attempts to advance the head by one node, and on success – it returns the oldest item to the caller. On failure it retries the whole scheme.

To signal to the recovery procedure that it should ignore the dequeued node, a successful dequeue operation ensures that the head's index is persistently increased to a value bigger than or equal to its dequeued node's index. Persisting the new head's index is intended to indicate to the recovery not only that this node is dequeued, but also that all nodes up to this one are dequeued, and a failing dequeue also needs to persist the head's index before returning in order to persist the previous dequeues that emptied the queue. This is obligatory due to the following observation:

Observation 6.5.2. The recovery must restore a consecutive prefix of dequeued nodes, to satisfy the FIFO order requirement.

[3] Applying Assumption 6.2.1 requires that the whole node resides on a single cache line, which is typically the case, and it also holds for the queues implemented in this work. The method of [CFL17] can be used to generalize the algorithms to nodes that span multiple cache lines without adding fence operations.

The recovery achieves this by interpreting nodes with index smaller than or equal to the head's index as dequeued.

A successful dequeue is responsible to reclaiming the node that was the head during the previous dequeue that this thread executed. This node to be retired is kept in a *nodeToRetire* array, consisting of a cell per thread. Its cells do not share cache lines to avoid false sharing. Each thread may access its cell using its thread ID as an index.

Next, we explain how dequeuers ensure that the correct head's index is restored by the recovery. If we let a dequeuer persist the head's address, and let the recovery determine the head's index to be the *index* in the node pointed to by this head (as appears in NVRAM in the crash moment), then the recovery might erroneously restore a stale (smaller) head's index value, and discard completed dequeues. This could happen if the enqueuer of the node pointed to by the head has linked the node but was interrupted by the crash before persisting the node's data. Therefore, UnlinkedQ takes a different approach to determine the head's index in recovery.

UnlinkedQ makes the head hold not only a pointer to the dummy node, but also its index. They are held side-by-side and updated together atomically using a double-width CAS. A dequeuer starts by performing a double-width CAS (Line 185) that advances the head's pointer and increments the head's index. Next, the dequeuer persists the index placed in the head (Line 187). A failing dequeue assists persisting the head's index too (Line 183).[4] The recovery procedure restores the head's index from the value kept in the queue's head, rather than from the possibly stale value in the node pointed to by the head. This prevents discarding a completed dequeue: persisting the head's index after incrementing it to the index of the dequeued node, makes the recovery procedure ignore the dequeued node.

The use of a double CAS can be eliminated (if the platform does not support it) by taking an alternative approach: Each thread could maintain a local index. After each time it advances the queue's head, it would update the local index with the value of the new head's index and persist it. The recovery would then restore the head's index as the maximum across these local indices. The alternative handling of the head's persistence described here, is actually required and applied in the second amendment of MSQ (see Section 6.6).

Recovery

The recovery procedure of UnlinkedQ resurrects nodes in the designated areas that are marked *linked* and have an *index* bigger than the head's index. It then sets their links to form a linked list that holds the queue nodes in the order induced by their indices. This is implemented as follows.

The head's index is not modified. A dummy node is allocated and assigned an index that matches the head's index. The head's pointer is set to point at this dummy node. Next, the recovery scans the designated areas and makes a list of recovered nodes, which are those

[4] We particularly specify the head's index as the flushed value in order to stress that this is the data required in recovery, but its flush clearly writes the whole containing cache line to the memory.

with a set *linked* flag and an *index* larger than the head's index. All other nodes are reclaimed. The recovered nodes are then sorted and their *next* pointers are set accordingly to create the queue. Finally, the queue's tail is set to point to the last node in the queue.

We note that free nodes (owned by the memory manager) in the designated areas are appropriately ignored by the recovery: When the memory manager allocates a new designated area for nodes from the operating system, it zeros its content, to make all nodes consist of a zeroed index, and then persists it in NVRAM (by placing asynchronous flushes of the whole area accompanied by a single SFENCE). If the number of required nodes is unknown in advance, each time a designated area is depleted, the memory manager may allocate a new area from the operating system and initialize it in a similar manner using a single SFENCE. The zeroed indices guarantee that the unused nodes owned by the memory manager are ignored by the recovery. In addition to these not-yet-allocated nodes, nodes reclaimed by dequeuers are also ignored by the recovery thanks to their index value, as dequeue operations return nodes to the memory manager only after the head's index persistently equals to the index of a subsequent node. Finally, nodes reclaimed by a previous recovery process are ignored thanks to either their *index* or their unset *linked*.

6.5.2 LinkedQ

LinkedQ also performs a single fence in each operation, but using a completely different approach. We start with an overview of LinkedQ. The first idea LinkedQ employs is to make the recovery procedure able to deal with nodes whose data has not been persisted. This allows linking nodes to the queue without blocking to persist their data beforehand, thus avoiding one of the fences of the queue in [FHMP18]. To enable this, LinkedQ presents a mechanism that identifies nodes with stale data: a designated *initialized* flag in each node signifies whether the content of the node is guaranteed to be valid. We maintain the invariant that if the node's data is not initialized in NVRAM, then its *initialized* flag is unset in NVRAM. To achieve this, LinkedQ's enqueue operation initializes the node in two steps: first, it initializes the node content, and then it sets the *initialized* flag. No SFENCE is issued during this execution, as Assumption 6.2.1 guarantees that the order of writes to the same cache line is not reversed.

For this scheme to work, we need to make sure that when a node is allocated, its *initialized* flag is unset. This can be easily done with an extra fence at allocation time, but would yield two fences per enqueue operation. We manage to avoid this fence by postponing the return of dequeued nodes to the memory manager. Think first of a simplified version that lets each thread accumulate k nodes it removed from the queue. After each k^{th} successful dequeue, before returning the k nodes to the memory manager, the thread clears their *initialized* flags, issues an (asynchronous) flush for each of the flags, and then a single blocking fence before letting the memory manager reclaim these objects. Such a simplified algorithm would execute $1+1/k$ fences per successful dequeue operation, not perfectly meeting the desired theoretical lower bound of a single fence. To reduce the number of fences to one, we take a more complex

approach: After removing a node N from the queue, its dequeueing thread T clears N's *initialized* flag and records N's address for later. Instead of placing an additional fence every k dequeues, T will piggyback on the fence which its next successful dequeue anyhow performs: T will flush N's *initialized* flag before this fence, and return N to the memory manager after this fence. Such piggybacking on a fence of a later operation by the same thread makes sure that *initialized* flags are properly reset in memory before their nodes are reused, without incurring additional fences.

The recovery procedure resurrects all nodes reachable from the head through a path of consecutive nodes with the *initialized* flag set. It remains to ensure that completed enqueue operations are visible to the recovery procedure, even though previous nodes[5] in the queue may have been enqueued by operations that have not yet completed. Before an enqueue operation completes, LinkedQ makes sure that all data on nodes from the head to the enqueued node is written back to the NVRAM. This guarantees that the recovery will reach the new node in its traversal from the head. Naively, before an enqueue operation completes, the enqueuer could traverse all nodes from the head until the new node, flush their contents, and then issue a single fence. This persists all relevant nodes but at a very high cost. To make this process efficient, we add a backward edge to the underlying linked list, and walk backwards persisting only nodes that might have not yet been persisted. We attempt to minimize the length of the walk as much as possible.

The LinkedQ algorithm appears in Figure 6.2. Next, we describe its operations in detail.

The Enqueue Operation

The enqueue operation first allocates a node denoted *newNode* from the memory manager and initializes its data (Lines 237–239). Then it sets *newNode*'s *initialized* flag (Line 240). In what follows, the enqueue operation attempts to link *newNode* to the last node (Line 245). Note that it might have just linked a node whose data is not persisted in NVRAM. If the link to *newNode* is written back to NVRAM (which could happen implicitly due to a cache eviction) and then a crash occurs, the recovery would have to deal with reaching a node with stale data. The correctness is maintained using the *initialized* flag and a matching recovery procedure: The *initialized* flag is used as a stamp indicating to the recovery that *newNode* is initialized. Relying on Assumption 6.2.1, the order of writing *initialized* after *newNode*'s data is preserved in NVRAM. Accordingly, the recovery resurrects only nodes with the *initialized* flag set. This guarantees that it resurrects only nodes with persisted data. If a crash occurs after the link to *newNode* is flushed to NVRAM and before *newNode*'s data is written back to NVRAM, then *newNode*'s *initialized* flag must be unset in NVRAM, thus, the recovery will ignore it (and all nodes linked after it).

The recovery procedure resurrects all nodes reachable from the head through a path of consecutive nodes with the *initialized* flag set. Since durable linearizability allows the recov-

[5] We think of the nodes as ordered by the underlying linked list of the queue. This order enables the terms *previous, preceding, subsequent, consecutive,* etc.

ery procedure to ignore enqueue operations that are concurrent with a crash and elide their nodes from the queue, the fact that the recovery procedure ignores nodes of ongoing enqueues that were linked after a node with an unset *initialized*, does not break durable linearizability. However, after an enqueue operation completes, the recovery procedure must not discard it, even if earlier nodes belong to incomplete enqueue operations. To this end, after successfully linking *newNode*, the enqueue operation ensures that the path of nodes leading from the head to *newNode* is persisted (Lines 246–247). This could be achieved naively by flushing all nodes from the head until *newNode*. To save redundant flushes, an enqueuer avoids flushing a prefix of queue's nodes that are guaranteed to be already flushed. Instead, it flushes only a suffix of queue's nodes that are not guaranteed to be persistent. To identify the relevant suffix, we place backward links in the nodes, but we remove a backward link when we know that all previous nodes in the queue have already been persisted. The backward links preserve the following invariant: all queue nodes (starting from the current queue's head) that precede a node with a nullified backward link have all relevant content (their item, set *initialized* flag and a non-NULL forward link) persisted.

```
207 class Node
208     Item* item
209     atomic⟨Node*⟩ next
210     atomic⟨Node*⟩ pred
211     bool initialized
```

```
212 Item* Dequeue()
213     while (true)
214         head = Head
215         headNext = head–>next
216         if (headNext == NULL)
217             FLUSH(&Head); SFENCE
218             return NULL
219         if (CAS(&Head, head, headNext)
220             dequeuedItem = headNext–>item
221             if (nodeToPersistAndRetire[tid])
222                 FLUSH(&
                        ↪ nodeToPersistAndRetire
                        ↪ [tid]–>initialized)
223             FLUSH(&Head)
224             SFENCE
225             headNext–>pred = NULL
226             if (nodeToPersistAndRetire[tid])
227                 retire(nodeToPersistAndRetire[
                        ↪ tid])
228             head–>initialized = false
229             nodeToPersistAndRetire[tid] =
                    ↪ head
230             return dequeuedItem
```

```
231 FlushNotPersistedSuffix(notPersisted)
232     do
233         FLUSH(notPersisted)
234         notPersisted = notPersisted–>pred
235     while (notPersisted != NULL);
236 Enqueue(item)
237     newNode = allocNode()
238     newNode–>item = item
239     newNode–>next = NULL
240     newNode–>initialized = true
241     while (true)
242         tail = Tail
243         if (tail–>next == NULL)
244             newNode–>pred = tail
245             if (CAS(&tail–>next, NULL,
                    ↪ newNode))
246                 FlushNotPersistedSuffix(
                        ↪ newNode)
247                 SFENCE
248                 CAS(&Tail, tail, newNode)
249                 // All nodes preceding
                        ↪ newNode are
                        ↪ persistent
250                 newNode–>pred = NULL
251                 break
252         CAS(&Tail, tail, tail–>next)
```

Figure 6.2: LinkedQ implementation

To maintain a backward path connecting the linked list's nodes that should be flushed, an enqueuer links a node with a backward link pointing to the previous node (Line 244). After linking *newNode*, its enqueuer traverses the queue from *newNode* backwards using the backward links, until reaching a NULL backward link, and flushes the content of all traversed nodes (including *newNode* itself) (Lines 232–235). Finally, it issues a single SFENCE to block until all these flushes complete (Line 247). By the above-mentioned invariant, all nodes starting from the current head and preceding this suffix of nodes, are persistent. Now, as this suffix is persisted as well, the data of all nodes preceding *newNode* starting from the current head is guaranteed to be persistent. As an optimization to prevent future enqueues from flushing these nodes, the enqueuer then sets *newNode*'s backward link to NULL (Line 250). Thus, each enqueue operation that reaches *newNode* from now on, during its backward walk, would not need to traverse the preceding persistent nodes. Note that backward links are not used in the recovery and there is no need to explicitly flush them.

To complete the enqueue operation, the tail is advanced to point to *newNode* (Line 248). Like in the original MSQ, a concurrent enqueue might prevent the enqueue's linking. In this case, the enqueuer tries to assist and advance the tail to point to the node enqueued by the obstructing enqueue (Line 252), before starting a new attempt to enqueue its own item.

We note that, as an optimization on x86 platforms, the SFENCE in Line 247 can be eliminated, because the following CAS instruction serves as an SFENCE guaranteeing completion of previous flushes. In the measured implementations of all algorithms, each SFENCE preceding a CAS is eliminated. We did not include this optimization in the thesis's pseudocode for clarity.

The Dequeue Operation

The dequeue operation attempts to extract the oldest item, placed in the node subsequent to the dummy node. If the queue is empty when the dequeue operation takes effect, it returns NULL. But before returning, the failing dequeue must persist the head (Line 217), to ensure that previous ongoing dequeues that emptied the queue are persistent. Otherwise, if a crash occurs after the failing dequeue returns, the previous dequeues might be discarded. This would break durable linearizability, since it will be impossible to linearize the completed failing dequeue correctly as applied to an empty queue, without the previous dequeues being linearized beforehand.

If the queue is not empty, the dequeuer attempts to advance the head by one node (Line 219), and on success – returns the oldest item to the caller. On failure it retries the whole scheme. Before returning, the dequeuer persists the head (Lines 223–224), to comply with durable linearizability, which requires that completed operations be linearized.

Each dequeue makes sure that the dummy node from which it advances the head will be unreachable by future operations, so that the next successful dequeue by the same thread will safely return this dummy node to the memory manager. To make it unreachable by backward walks (of enqueue operations that will try to identify a not persisted suffix), the dequeuer

disconnects the backward link from the new dummy head to the previous one (Line 225). In addition, persisting the head guarantees that the previous dummy node will be unreachable by future operations even in case of a crash.

A successful dequeue does not simply return the previous dummy node (i.e., the node from which the previous successful dequeue by the same thread has advanced the head) to the memory manager as it is. Recall from Section 6.5.2 that we must make sure that newly allocated nodes have their *initialized* flags reset. The *initialized* flag placed in each node is used by its enqueuer to signal to the recovery when the node's data is persisted. Suppose a node is erroneously allocated in an enqueue operation with a set *initialized* flag. After the enqueue operation links the node, the link to the node might be implicitly flushed to the NVRAM, and – before the node's data is persisted – a crash might follow. The recovery procedure would then find the linked node, containing stale data including a set *initialized* flag, and would erroneously interpret the node with the stale content as part of the queue. To prevent this scenario, enqueuers could unset the *initialized* flag after the node's allocation and then persist it before initializing its data, but this incurs an additional fence. Instead, we make sure that a node is always allocated with an *initialized* flag persistently unset. Next we explain how we ensure that.

If we allocate nodes from the operating system, we would get nodes with arbitrary content, possibly with the *initialized* field set. Instead, we implement a memory manager that maintains large designated areas from which all node allocations are performed.

First, we explain how nodes, allocated from the designated areas for the first time, are allocated with a persistently unset *initialized* value. If the number of nodes required by the program is known in advance, then on program startup, the memory manager may allocate a sufficiently large designated area for nodes from the operating system, zero its content to make all nodes marked as not initialized, and then persist it in NVRAM (by placing asynchronous flushes of the whole area accompanied by a single SFENCE). This guarantees that when the memory manager allocates a node for the first time, its *initialized* field is unset. If the number of required nodes is unknown in advance, each time a designated area is depleted, the memory manager may allocate a new area from the operating system, and initialize it in a similar manner using a single SFENCE.

It remains to explain how nodes, reallocated from the designated areas after reclamation, are allocated with an *initialized* flag persistently unset. The dequeue operation and the recovery procedure return nodes to the memory manager, hence, they are responsible to return them with an *initialized* flag persistently unset.

Starting with dequeue, a successful dequeuer could unset the *initialized* flag of the dummy node from which it has advanced the head and then perform additional flush and SFENCE to persist the unset *initialized* flag before returning the node to the memory manager. However, to achieve the fence lower bound of a single SFENCE per operation, LinkedQ takes a different approach.

The persistence of the previous dummy node's *initialized* flag is accomplished through piggybacking on the next successful dequeue's SFENCE, which this thread is anyhow go-

ing to execute (in Line 224). More precisely, the dequeuer sets the previous dummy node's *initialized* flag to *false* (Line 228) after the queue's head persistently points to a subsequent node. The dequeuer thread postpones the reclamation of this previous dummy node, and keeps the node locally in a *nodeToPersistAndRetire* array (Line 229). This array consists of a pointer cell per thread, each cell lying in another cache line to avoid false sharing. Each thread may access its cell using its thread ID as an index. In the next successful dequeue execution of the same thread, right before its SFENCE, the *initialized* flag of the node we kept aside is flushed (Line 222). After the fence completes, the node may be returned to the memory manager (Line 227).

As for the recovery, as detailed in Section 6.5.2, for each node with a set *initialized* flag that it returns to the memory manager – the recovery unsets the flag and flushes it. A single SFENCE placed in the end of the recovery ensures that these flags are unset in the memory.

Recovery

The recovery procedure of LinkedQ, running after a crash, resurrects all nodes reachable from the head through a path of consecutive nodes with the *initialized* flag set. It does so by leaving the queue's head as it is and reconstructing the queue as follows.

1. If the *initialized* flag of the dummy node (namely, the node pointed to by the head) is unset, the recovery procedure sets the dummy node's *next* to NULL and then sets its *initialized* flag. The order of the last two writes ensures (based on Assumption 6.2.1) a proper recovery from a possible crash in the midst of the current recovery. The tail is set to point to the dummy node as well.

2. Otherwise (the dummy node's *initialized* is set) –

   (a) The recovery procedure traverses the nodes starting with the one pointed to by the head, until it reaches either a node whose *next* value is NULL, or a node with an unset *initialized*. In the first case, the recovery points the queue's tail to the last traversed node.

   (b) If the traversal ends due to a node with an unset *initialized* flag, then let P be its preceding node. The recovery sets P.*next* to NULL and flushes it, and sets the tail to point to P.

In all cases, the *pred* field of the last node (pointed to by the tail) is set to NULL. In addition, throughout the queue traversal, the addresses of all traversed nodes with a set *initialized* flag are recorded. All other nodes in the designated allocation areas are reclaimed. For each reclaimed node with a set *initialized* flag, the recovery unsets the *initialized* flag and flushes it before retiring the node. There could be at most two such nodes per thread: There is at most one such node (namely, a node which is not part of the queue but has a set *initialized* flag) which the thread has dequeued and placed in its local *nodeToPersistAndRetire* array, where the node awaits its persistence. In addition, there could be another such node – a node that the

thread was about to enqueue, if the thread were in the middle of an enqueue operation when the crash occurred; or alternately a node that the thread has just advanced the head from, if the thread were in the middle of a dequeue operation when the crash occurred.

If any flush were executed during the recovery, a single SFENCE is placed in the end to ensure the completion of the executed flushes.

6.6 Second Amendment: Queues with No Post-Flush Access

It turns out in the evaluation that reducing the number of fences is not enough to obtain high performance, and one should further improve the algorithms by reducing accesses to flushed data. In this section we describe further transformations of UnlinkedQ and LinkedQ into the new algorithms OptUnlinkedQ and OptLinkedQ respectively which do not access flushed locations, while still executing the minimal possible number of blocking fences per operation. Evaluation will show that the obtained algorithms yield excellent performance in current architectures. These algorithms are the fastest available persistent queues today, but we believe that UnlinkedQ and LinkedQ are of value on their own. This is because future architectures may provide flushes that do not invalidate cache lines. In such architectures UnlinkedQ and LinkedQ are expected to perform well thanks to using the minimal number of fence instructions. However, we cannot evaluate this performance prediction on the platform we currently possess.

6.6.1 OptUnlinkedQ

We start with looking at what data is flushed in the UnlinkedQ algorithm, for use in a recovery. UnlinkedQ flushes the global head index, plus, the *index*, *item* and *linked* fields for each node in the underlying linked list. All of these values except for the *linked* field are later accessed. We eliminate these accesses using algorithmic modifications, amending UnlinkedQ to become OptUnlinkedQ.

First, we switch the global head index with a per-thread head index, holding the value that the head index had during the last dequeue by the thread. In OptUnlinkedQ the head pointer is a pointer only (with no adjacent index). Instead of persisting the global head index in the end of every dequeue operation as UnlinkedQ does, a dequeuer of OptUnlinkedQ copies the index value of the node pointed to by the head pointer to its local head index and persists it. In a recovery, the head index is set to the maximal index among the local head indices of all threads. Note that in this description we write to the local head index after persisting it. We eliminate this access in Section 6.6.3 below.

The *index* and *item* fields of a node in UnlinkedQ are written by the node's enqueuer, and then (after the node is linked to the queue) – flushed by it, as well as read by subsequent operations: the *item* is read by a subsequent dequeue and the *index* is read by subsequent enqueuers. To prevent reads of a location after it is flushed, an enqueuer in OptUnlinkedQ physically splits the node into two nodes. The first one is called Persistent and it is flushed and

not accessed after the flush. It is only used during a recovery, for which its content is essential. It is allocated in the designated areas that the recovery will scan. The second node is denoted Volatile and it is not flushed and not used in a recovery. However, Volatile is accessed after the flush of Persistent and is utilized to expedite the normal operation on the object. The *index* and *item* fields are placed in both Persistent and Volatile, with the two copies of each of them set to the same value. The enqueuer persists Persistent, while subsequent operations read the *index* and *item* from Volatile, thus adhering to our guideline. To enable access to the non-flushed fields, the queue's head and tail point to the Volatile part.

Each part of the node contains additional fields other than *index* and *item*: The *linked* field is not accessed after the enqueuer performs the flush (except for during recovery), so there is no need to keep two copies of it, and it is placed in Persistent only. The two following additional fields, which are not required in recovery, are placed in Volatile: *next*, and a pointer to the associated Persistent object, which the enqueuer sets for enabling the thread that reclaims the node later to locate Persistent and reclaim it together with Volatile.

The recovery procedure of OptUnlinkedQ resurrects Persistent objects in the designated areas that are marked *linked* and have an *index* bigger than the head's index. It then allocates matching Volatile objects and links them in a linked list in the order induced by their indices. This is implemented as follows.

Let *headIndex* be the maximal index among the local head indices of all threads. These per-thread indices are not modified. A dummy Persistent object is allocated and assigned the index *headIndex*. Next, the recovery scans the designated areas and makes a list of recovered Persistent objects, which are those with a set *linked* flag and an *index* larger than *headIndex*. All other Persistent objects are reclaimed. Then, in order to construct a queue of Volatile objects, for each of the recovered Persistent objects, as well as for the dummy Persistent, the recovery allocates a Volatile object and sets a pointer from it to the associated Persistent object. In addition, the *index* and *item* of each Volatile are copied from the associated Persistent. The Volatile objects are sorted by their indices, and their *next* pointers are set accordingly to create the queue. Finally, the queue's head and tail pointers are pointed at the first and last Volatile objects in the linked list.

Figure 6.3 contains the pseudocode of the OptUnlinkedQ algorithm. Note that the queue's global head and tail pointers point to Volatile nodes. The MOVNTI instruction is a non-temporal store instruction that writes back data directly to the memory, bypassing the caches.

6.6.2 OptLinkedQ

Transforming LinkedQ to a queue with no access to flushed data is trickier and involves further modifications, since it is problematic to eliminate accesses to a node's *next* field after its flush. It is easier to avoid accessing a node's backward link *pred* after its flush, so we make the recovery rely on the node's *pred* instead of *next*. Accordingly, the recovery mechanism is reversed, so that instead of resurrecting a path of consecutive valid nodes reachable from the head (as LinkedQ does), OptLinkedQ resurrects a chain of consecutive nodes reachable

Figure 6.3: OptUnlinkedQ implementation

```
253 class Persistent
254     Item* item
255     int index
256     bool linked
257 class Volatile
258     Item* item
259     int index
260     atomic⟨Volatile*⟩ next
261     Persistent* persistentNode
```

```
262 Item* Dequeue()
263     while (true)
264         head = Head
265         headNext = head->next
266         if (headNext == NULL)
267             movnti(&localData[tid].headIndex,
                ↪ head->index)
268             SFENCE
269             return NULL
270         if (CAS(&Head, head, headNext)
271             dequeuedItem = headNext->item
272             movnti(&localData[tid].headIndex,
                ↪ headNext->index)
273             SFENCE
274             if (localData[tid].nodeToRetire)
275                 retire(localData[tid].
                    ↪ nodeToRetire
                    ↪ ->persistentNode)
276                 retire(localData[tid].
                    ↪ nodeToRetire)
277             localData[tid].nodeToRetire = head
278             return dequeuedItem
```

```
279 Enqueue(item)
280     newNode = allocVolatile()
281     newNode->item = item
282     newNode->next = NULL
283     newNode->persistent = allocPersistent()
284     newNode->persistent->item = item
285     newNode->persistent->linked = false
286     while (true)
287         tail = Tail
288         if (tail->next == NULL)
289             newNode->persistentNode->index
                ↪ = tail->index + 1
290             newNode->index = newNode
                ↪ ->persistentNode->index
291             if (CAS(&tail->next, NULL,
                ↪ newNode))
292                 newNode->persistentNode
                    ↪ ->linked = true
293                 FLUSH(newNode
                    ↪ ->persistentNode);
                    ↪ SFENCE
294                 CAS(&Tail, tail, newNode)
295                 break
296         CAS(&Tail, tail, tail->next)
```

from the tail by backward links, ending with the node succeeding the dummy node. Similarly to OptUnlinkedQ, the queue node will be split into two nodes (Persistent and Volatile) so that the fields accessed after a flush (including the forward links) will not reside on the same cache line with the flushed fields (including the backward links).

Maintaining a single fence in each enqueue operation complicates the design of OptLinkedQ further: An enqueuer needs to use a single fence to ensure the persistence of both all recently inserted nodes and the tail. Therefore, before the final fence, the tail might be already persisted while some nodes are not, which may cause the recovery to encounter stale nodes when walking from the tail backwards. The way we deal with this problem is to let the recovery identify stale nodes during the traversal. When a stale node is discovered, the recovery starts over from an older recorded value of the tail, and repeats this process until finding a recorded tail value from which the node succeeding the head is reachable through a chain of persisted nodes. An *index* field placed in the nodes allows the recovery to identify

stale nodes. These are nodes whose *index* value is nonconsecutive. This field is set in a new node by its enqueuer, to the index of the last node plus 1.

The *index* field in nodes is also utilized to recover the head and the tail. As for the head, we cannot let dequeues flush the head pointer, because it will be accessed thereafter by following dequeues. Like in OptUnlinkedQ, we assign a per-thread head index, which dequeues update with the head index and persist, and recover the head index as the maximum among these values in all threads. The recovery terminates its backward walk when it reaches a node with the head index plus 1.

We can also not let enqueues flush the tail, because it will be accessed thereafter by subsequent enqueues. To solve this, we assign a per-thread last-enqueue pointer (pointing to the last Persistent object enqueued by the thread), as well as a per-thread last-enqueue index. Note that a backward walk from a last-enqueue pointer of a thread that performed an enqueue during the crash, might pass through stale nodes, as the per-thread last-enqueue pointer and index might be persisted before some queue nodes are persisted. Thus, the recovery looks for the per-thread last-enqueue pointer pointing to the latest node *up to which all nodes are persisted*. The recovery starts the traversal from the node pointed to by the per-thread last-enqueue pointer with the maximum associated per-thread last-enqueue index among all threads, and if the *index* of this node is different from the associated last-enqueue index, or if nonconsecutive *index* values are encountered (each of these cases implies that the inspected node is stale), it restarts the walk from the next last-enqueue pointer candidate, which is the one with the next largest associated index, until it identifies a Persistent object from which it establishes a complete walk up to the node succeeding the head.

The recovery scheme cannot be complete without dealing with the following rare scenario. All threads execute enqueues concurrently, the new last-enqueue pointer and index of them all are persisted in the memory, but then a crash occurs before any of the new nodes is persisted. In such a case, all last-enqueue pointers in all threads point to stale nodes, and the recovery will identify them as such. To restore a valid tail in this case, we assign *two* per-thread last-enqueue pointer and index, in which each thread keeps the details of both the last node enqueued by this thread and the penultimate node enqueued by this thread (up to which all queue nodes are definitely persisted by now because the penultimate enqueue was completed, including its fence instruction). The recovery sorts all last-enqueue indices (two of each thread) from largest to smallest and gathers their matching pointers to a single list of potential tail pointers. It attempts starting a backward walk from them, one after another. For each attempted tail pointer, if the index in the node it points to is different from the associated local enqueue index, or if a nonconsecutive index is encountered during the backward walk from it to the node with the recovered head index plus 1 (each of these cases implies that the index of the inspected node is stale) – the recovery moves on to try the next potential tail.

An enqueuer sets the *index* of the new node after setting its *item* and *pred*, so based on Assumption 6.2.1, when the recovery identifies the node's *index* as non-stale, it is guaranteed that its *item* and *pred* values are not stale. In this new recovery scheme that uses *index* to detect stale nodes, an *initialized* field like in LinkedQ is redundant.

Overall, the node's fields required in the recovery of OptLinkedQ are *index*, *item* and *pred*. A node of OptLinkedQ is composed of the following two parts: Persistent consists of the above mentioned fields, and Volatile consists copies of these fields for access after the flush of Persistent, as well as a *next* field that is not required in recovery, and a pointer to the associated Persistent object for its later reclamation.

OptLinkedQ Details

The pseudocode of the OptLinkedQ algorithm appears in Figures 6.4 and 6.5. The queue's global head and tail pointers point to Volatile nodes. *localData* is an array consisting of a cell per thread. Each thread may access its cell using its thread ID as an index. Each cell consists of the fields *headIndex* and *nodeToRetire* accessed in dequeues, and *lastEnqueues* (an array containing two cells, each composed of a pointer to a Persistent object and an index), *lastEnqueuesIndex* and *validBit* accessed in enqueues. *localData* array's cells do not share cache lines to avoid false sharing. In addition, for each cell, the *lastEnqueues* array and *headIndex* field, which are written using MOVNTI instructions, are kept in a cache line separate from the rest of the cell's fields.

Next, we describe OptLinkedQ's operations in detail.

The Enqueue Operation The enqueue operation first allocates a Volatile node denoted *newNode* from the memory manager and a matching Persistent node and initializes their data (Lines 343–347). Then, before attempting to link *newNode* to the last node, it sets the *pred* and *index* fields of both the Volatile and Persistent parts (Lines 351–354). The *index* field of the Persistent object serves as a stamp indicating to the recovery that the object's data is up-to-date: *index* is the last written field of the Persistent object, for ensuring that if this object is traversed during a recovery walk, and its *index* is identified as non-stale, then all the object's data is non-stale. This is due to Assumption 6.2.1, guaranteeing that the order of writing *index* after the other fields is preserved in NVRAM.

Next, the enqueue operation attempts to link *newNode* to the last Volatile node (Line 355), and on success it advances the queue's tail and ensures that the path of nodes leading from the head to *newNode->persistentNode* is flushed to the NVRAM (Lines 356–357).

It then records the address and index of the newly enqueued Persistent node in the thread's *lastEnqueues* array (Line 358). This array contains two cells per thread – for keeping record of the thread's last and penultimate enqueued nodes. The thread writes alternately – on each enqueue it writes to the cell with index *localData[tid].lastEnqueueIndex* and in the end flips its *lastEnqueueIndex* (in Line 341). The writes to *lastEnqueues* are performed using MOVNTI instructions (Lines 338–339). In case a crash occurs after only one of the address and index was written to the memory, the subsequent recovery needs to identify that the cell's content is invalid and should be ignored. To this end, we place a valid bit in both the address and value (the least significant bit of the address and the most significant bit of the index). A *lastEnqueues* cell is considered valid only if the valid bits of its address and index match.

After the writes, the value of *localData[tid].validBit* is flipped if *localData[tid].lastEnqueues=1* (Line 340), so that the thread's following writes to its two *lastEnqueues* cells will be with the opposite valid bit value.

Finally, the enqueue operation issues an SFENCE (Line 359) to ensure the completion of all executed flushes and MOVNTI instructions. In particular, all Persistent nodes succeeding the current head up to *newNode->persistentNode* are guaranteed to be persistent. To prevent future enqueues from redundantly flushing these nodes, the enqueuer then sets *newNode*'s backward link to NULL (Line 361). Thus, each enqueue operation that reaches *newNode* from now on, during its backward walk, would not need to traverse the preceding Persistent nodes.

Like in the original MSQ, a concurrent enqueue might prevent the enqueue's linking. In this case, the enqueuer tries to assist the obstructing enqueue and advance the tail to point to the node enqueued by that obstructing enqueue (Line 363), before starting a new attempt to enqueue its own item.

The Dequeue Operation The dequeue operation attempts to extract the oldest item, placed in the node subsequent to the dummy node. If the queue is empty when the dequeue operation takes effect, it returns NULL. But before returning, the failing dequeue must ensure that previous dequeues that emptied the queue survive a crash. It does so by copying the head's index to its local head index and persisting it (Lines 312–313). Each thread's local head index variable is placed in the thread's cell in the *localData* array.

If the queue is not empty, the dequeuer attempts to advance the head by one node (Line 315),

```
297 class Persistent
298     Item* item
299     Persistent* pred
300     int index
301 class Volatile
302     Item* item
303     atomic⟨Volatile*⟩ next
304     atomic⟨Volatile*⟩ pred
305     int index
306     Persistent* persistentNode
```

```
307 Item* Dequeue()
308     while (true)
309         head = Head
310         headNext = head->next
311         if (headNext == NULL)
312             movnti(&localData[tid].headIndex,
                    ↪ head->index)
313             SFENCE
314             return NULL
315         if (CAS(&Head, head, headNext)
316             dequeuedItem = headNext->item
317             movnti(&localData[tid].headIndex,
                    ↪ headNext->index)
318             SFENCE
319             headNext->pred = NULL
320             if (localData[tid].nodeToRetire)
321                 retire(localData[tid].
                        ↪ nodeToRetire
                        ↪ ->persistentNode)
322                 retire(localData[tid].
                        ↪ nodeToRetire)
323             localData[tid].nodeToRetire = head
324             return dequeuedItem
```

Figure 6.4: OptLinkedQ implementation – Objects and Dequeue

Figure 6.5: OptLinkedQ implementation – Enqueue

```
325 FlushNotPersistedSuffix(notPersisted)
326     while (true)
327         pred = notPersisted->pred
328         if (pred == NULL)
329             break
330         FLUSH(notPersisted->persistentNode)
331         notPersisted = pred
332 ZeroBit(value, bitIndex)
333     return value & ~(1 ⟨⟨bitIndex)
334 ApplyBit(value, bitIndex, bitValue)
335     return ZeroBit(value, bitIndex) | (bitValue
            ↪ ⟨⟨bitIndex)
336 RecordLastEnqueue(newNode)
337     i = localData[tid].lastEnqueuesIndex
338     movnti(&localData[tid].lastEnqueues[i].ptr,
            ↪ ApplyBit(newNode
            ↪ ->persistentNode, 0, localData[tid
            ↪ ].validBit))
339     movnti(&localData[tid].lastEnqueues[i].
            ↪ index, ApplyBit(newNode->index,
            ↪  sizeof(newNode->index)*8-1,
            ↪ localData[tid].validBit))
340     localData[tid].validBit ^= i // Flip valid
            ↪ bit if i=1
341     localData[tid].lastEnqueuesIndex ^= 1 //
            ↪ Flip index
```

```
342 Enqueue(item)
343     newNode = allocVolatile()
344     newNode->item = item
345     newNode->next = NULL
346     newNode->persistentNode =
            ↪ allocPersistent()
347     newNode->persistentNode->item = item
348     while (true)
349         tail = Tail
350         if (tail->next == NULL)
351             newNode->pred = tail
352             newNode->index = tail->index +
                    ↪ 1
353             newNode->persistentNode->pred
                    ↪ = tail->persistentNode
354             newNode->persistentNode->index
                    ↪ = newNode->index
355             if (CAS(&tail->next, NULL,
                    ↪ newNode))
356                 CAS(&Tail, tail, newNode)
357                 FlushNotPersistedSuffix(
                        ↪ newNode)
358                 RecordLastEnqueue(newNode)
359                 SFENCE
360                 // All nodes up to newNode
                        ↪ are persistent
361                 newNode->pred = NULL
362                 break
363         CAS(&Tail, tail, tail->next)
```

and on success – returns the oldest item to the caller. On failure it retries the whole scheme. Before returning, the dequeuer copies the new head's index to its local head index and persists it (Lines 317–318), to comply with durable linearizability, which requires that completed operations be linearized.

A successful dequeue is responsible for reclaiming the dummy node recorded by the previous dequeue executed by the same thread. Before reclaiming, it must ensure that the node is unreachable by future operations. To make it unreachable by backward walks (of enqueue operations that will try to identify a non-persisted suffix), the dequeuer disconnects the backward link from the new dummy head to the previous one (Line 319). It then returns the Persistent and Volatile objects of the previous dummy node to the memory manager (Lines 320–322), and keeps a record of the current dummy node for its future reclamation (Line 323).

Recovery The recovery procedure of OptLinkedQ resurrects all nodes reachable through backward links from the abstract tail until the node succeeding the dummy head. It then allocates matching Volatile objects and sets their forward links to form the linked list that constitutes the volatile queue. This is implemented as follows.

Let *headIndex* be the maximal index among the local head indices of all threads. The

recovery does not modify these per-thread indices. It sorts all per-thread *lastEnqueues*'s indices that are valid (namely, their valid bit value matches the valid bit value of the associated pointer), bigger than *headIndex* and have an associated non-NULL pointer from largest to smallest, and gathers them with their matching per-thread last enqueue pointers to a single list of potential tails. The recovery then attempts to start a backward walk from each potential pointer, one after another. For each attempted pointer, if the index in the Persistent object it points to is different from the associated index kept in the appropriate *lastEnqueues* cell, or if a nonconsecutive index is encountered during the backward walk from it to the Persistent object with index *headIndex+1* (each of these cases implies that the index of the inspected Persistent object is stale) – the recovery moves on to try the next potential tail.

All Persistent objects in the designated allocation areas but the ones traversed in the last successful walk are reclaimed (if there was such a walk, otherwise the queue is empty and all Persistent objects are reclaimed). For each reclaimed node with an index bigger than *headIndex* (there could be at most one such node per thread – for threads that were in the middle of enqueuing when the crash occurred), the recovery zeroes the node's index and flushes it before retiring the node.

In order to construct a linked list of Volatile objects, for each of the recovered Persistent objects, the recovery allocates a Volatile object and sets its Persistent pointer to the associated Persistent object. In addition, the *index* and *item* of each Volatile are copied from the associated Persistent. The *next* pointers of the Volatile objects are set according to the queue's order. The *pred* field of the last Volatile object is set to NULL. Dummy Volatile and Persistent objects are allocated too. Their *index* fields are set to *headIndex*. The Persistent pointer of the dummy Volatile object is pointed at the dummy Persistent object. The *next* pointer of the dummy Volatile is pointed at the recovered Volatile object with index *headIndex+1*, or set to NULL if an empty queue is recovered. The queue's head and tail pointers are pointed at the first and last Volatile objects in the linked list respectively.

For all threads that do not contain a valid record of the recovered tail in any of their *lastEnqueues* cells, these cells are zeroed using MOVNTI instructions. In addition, their *lastEnqueuesIndex* is set to 0, and their *validBit* is set to 1. For a thread with a valid *lastEnqueues* cell referring to the recovered tail: Its other cell is zeroed using MOVNTI instructions. In addition, its *lastEnqueuesIndex* is set to the other cell's index, and the thread's *validBit* is set appropriately (so that the next write to the cell that refers to the recovered tail will be with a valid bit value opposite of its current one).

Finally, the recovery issues an SFENCE to ensure the completion of all executed flushes and MOVNTI instructions.

6.6.3 Direct Write-Backs to Memory

The scheme described for OptUnlinkedQ replaces the global head index of UnlinkedQ, which is read, written and persisted for an unbounded number of times, with local variables that are never read (except for during recovery). However, they are still written and persisted for

an unbounded number of times: each dequeue operation writes and persists the local head index of its thread. A standard write to a value that is absent from the cache causes a fetch of the containing cache line from the memory. Thus, we wish to avoid such a write to a flushed (thus evicted) location. Instead of a standard write, we issue a non-temporal write (using the MOVNTI instruction) of the local head index, which writes back the value to the memory without touching the cache. This way, OptUnlinkedQ optimally performs no access to flushed cache lines.

To achieve this goal for OptLinkedQ as well, we need to eliminate any access to its local variables. The head index is handled just like in OptUnlinkedQ, using non-temporal writes. In addition, the local last-enqueue pointers and indices are also written and persisted for an unbounded number of times, and we update them too using non-temporal writes.

6.7 Durable Linearizability

To define linearization points for our queue algorithms, we first define some supporting terminology. We start with *volatile linearization points*, which match the standard linearization points of MSQ, and are intuitively the steps applying the operations to the volatile queue. We also define a *survival point* for each operation, which marks the time from which the operation survives a crash. These two terms should basically be interpreted as: if an operation passes its survival point, then it is linearized at the time of its volatile linearization point. If it does not reach its survival point, then it is not linearized in this execution. Then, we derive the abstract state of the queue for each possible state of the queue's underlying list of nodes.

6.7.1 Linearization Points

Definition 6.7.1 (Volatile Linearization Point)**.** For each operation *op* in an execution *E* of the queue, we define its *volatile linearization point* to be the same as *op*'s standard linearization point in MSQ:

- Enqueue's volatile linearization point is the CAS that links its new node (Volatile object in case of OptUnlinkedQ and OptLinkedQ) to the last one.
- For a successful dequeue, its successful CAS that advances the queue's head is its volatile linearization point.
- The volatile linearization point of a failing dequeue is reading the *next* pointer of the dummy node (Volatile object in case of OptUnlinkedQ and OptLinkedQ), which is later revealed to be NULL.

An operation in *E* that does not reach its volatile linearization point as defined above, does not have a volatile linearization point (similarly to how not all operations in an execution have a linearization point). Intuitively, an operation's volatile linearization point is the step that applies the operation to the volatile queue.

Definition 6.7.2 (Survival Point)**.** For each operation *op* in an execution E of the queue, we define a *survival point* as follows:

- **Successful Dequeue.** Let *op* be a successful dequeue that advances the head to point to *N* at moment t.

  op's survival point in UnlinkedQ and LinkedQ is the first (implicit or explicit) flush of the queue's head to the NVRAM after t, if a crash does not happen between t and this flush (else, the dequeue operation does not have a survival point). (Note that the flushed value of the head could be pointing to *N* or a subsequent node in the queue).

  op's survival point in OptUnlinkedQ and OptLinkedQ is the first (implicit or explicit) flush of a per-thread head index to the NVRAM after t with a value greater than or equal to *N*'s index, if a crash does not happen between t and this flush (else, the dequeue operation does not have a survival point).

- **Failing Dequeue.** Let *op* be a failing dequeue. Let *head* be the last value read off the queue's head, before discovering the queue is empty. This read is followed by *op*'s volatile linearization point, where the *next* pointer in *head* is read and found NULL. Let t_ℓ be the time of this volatile linearization point, and let us look back in time at the point t where the value *head* was written to the queue's head. Let t_p be the first time after t, where the content of the queue's head was flushed (implicitly or explicitly) to the memory, if a crash does not happen between t and this flush (else, t_p is undefined, and so is the survival point of the dequeue). Then *op*'s survival point in UnlinkedQ and LinkedQ is defined to be the later between t_ℓ and t_p.

  op's survival point in OptUnlinkedQ and OptLinkedQ is defined similarly but using an alternative definition of t_p, as the moment of the first (implicit or explicit) flush of a per-thread head index to the NVRAM after t with a value greater than or equal to *head->index*, if a crash does not happen between t and this flush (else, t_p is undefined, and so is the survival point of the dequeue).

- **Enqueue.** Let *op* be an enqueue operation that inserts *N* to the queue. By *N* we refer to a Node object linked to the queue in case of UnlinkedQ and LinkedQ, and to a Persistent object pointed to by a Volatile object that is linked to the queue in case of OptUnlinkedQ and OptLinkedQ. Then the first of the following events to occur in E after the linking and before a crash occurs, is *op*'s survival point (if none of the following happens after the linking and before a crash, then the enqueue operation does not have a survival point):

  1. The queues differ in this event:
     - For UnlinkedQ and OptUnlinkedQ: An (implicit or explicit) flush of *N*'s *linked* field to the NVRAM after it is set to *true*.
     - For LinkedQ: The first time when all of the following conditions have been met, for some node preceding *N*, denoted *dummy* (intuitively, *N* has become

reachable from *dummy* and marked as initialized in the NVRAM view):

(a) The queue's head has been flushed (implicitly or explicitly) to the NVRAM with a pointer to *dummy*. (Intuitively: *dummy* has become the queue's dummy node in the NVRAM view.)

(b) The underlying linked list of the queue connects *dummy* to *N*; and for each of the nodes along the way excluding *N*, its *next* field pointing to the subsequent node has been flushed (implicitly or explicitly) to the NVRAM. (Intuitively: *N* has been linked to the queue in the NVRAM view.)

(c) The setting of a *true* value to the *initialized* field in *N* reaches NVRAM by an (implicit or explicit) flush of *N*. (Intuitively: *N* has been marked as valid in the NVRAM.)

– For OptLinkedQ: The first time when all of the following conditions have been met, for some Persistent object preceding *N*, denoted *dummy*, and some Persistent object denoted *last* that is either *N* or a later Persistent object (intuitively, a backward path from the tail to the head through *N* became persistent):

(a) Some per-thread head index has been flushed (implicitly or explicitly) to the NVRAM with the index of *dummy* (which means the head index will be recovered as *dummy*'s index or a bigger value).

(b) A last-enqueue pointer of some thread has been flushed (implicitly or explicitly) to the NVRAM with a pointer to *last*, and the associated last-enqueue index of that thread has been flushed to the NVRAM with the value *last.index*. (Intuitively: *last* has become a potential tail for the recovery.)

(c) The index of each Persistent object, from *last* backwards up to *dummy* excluding *dummy*, has been flushed (implicitly or explicitly) to the NVRAM with its final value (namely, the indices of all these Persistent objects have been flushed with consecutive values).

2. The survival point of a successful dequeue operation that dequeues the value inside *N*.

An operation in *E* that does not reach its defined-above survival point (in particular, a failing dequeue that does not reach both t_ℓ and t_p, and an enqueue that does not reach any of the two detailed points), either due to a crash or since the execution ends, does not have a survival point.

Intuitively, an operation's survival point is the flush that makes the operation survive a crash. The failing dequeue is somewhat different, as this operation does not modify the queue and we sometimes let its survival point be set to its volatile linearization point, rather than a flush. Operations that reach a survival point are linearized even if a crash occurs after their survival point before they complete. Note that for our queues the survival point always happens when the volatile linearization point has already occurred.

Definition 6.7.3 (Linearization Point)**.** The *linearization point* of an operation *op* in an execution *E* of the queue, is defined to be its volatile linearization point if *op* reaches a survival point in *E*. In this case, we say that *op* is *linearized*. Otherwise, *op* is not linearized, i.e., has no linearization point.

6.7.2 The Abstract State of the Queue

We define the abstract state of the queue at each moment (including during the recovery) in a given execution of each queue. This state reflects the applying of all operations linearized so far in their linearization order.

UnlinkedQ

The abstract head index in an execution of UnlinkedQ is set to the value[6] of the *index* field in the queue's head except in an interval before a crash. Between the last flush of the head to the NVRAM before a crash and the crash, the value of the abstract head index is not modified. It remains the value that was flushed to the memory.

The abstract state of the queue for execution E at moment t is defined as all items in nodes with indices bigger than the current abstract head index, which were enqueued by linearized enqueues whose linearization points occurred prior to t, ordered by their enqueues' linearization order.

LinkedQ

The abstract head of the queue in an execution E of LinkedQ is defined similarly to the abstract head defined for UnlinkedQ, but this time we look at the head pointer. We define the abstract head to be the queue's head value, except in an interval before a crash. From the last flush of the head (explicitly or implicitly) to the memory before a crash, until the crash, the abstract state of the head keeps the value flushed to the memory with no further abstract head state modifications in this interval.

Consider an execution E of the queue and a moment t during the execution, and consider the sequence of underlying list's nodes, starting with the dummy node pointed to by the abstract head, and ending with the first node along the chain whose *next* pointer is NULL or points to a node enqueued by a non linearized enqueue. Namely, we do not include nodes whose enqueues have not been linearized yet. The abstract state of the queue for E at t is the sequence of items contained in all these nodes except for the first one (the dummy node). Note that the abstract state of the queue is an empty sequence if and only if the *next* pointer of the dummy node is NULL or points to a node enqueued by a non linearized enqueue.

[6] To avoid confusion between the value in cache and the value in memory, we clarify that whenever a variable's value is mentioned, we refer to the last value written to the variable (regardless of whether it has reached the NVRAM).

OptUnlinkedQ

The abstract head index of the queue in an execution E of OptUnlinkedQ is set to the value of the *index* field in the node pointed to by the queue's shared head, except in an interval enclosing a crash. Let *headIndex* be the biggest per-thread head index value flushed (explicitly or implicitly) to the NVRAM before the crash. Between the moment a pointer to a node with the index *headIndex* is written to the queue's head and the moment the recovery procedure (that runs after the crash) terminates, the abstract head index keeps the value *headIndex*.

The abstract state of the queue for execution E at moment t is defined as all items in Persistent objects with indices bigger than the current abstract head index, which were enqueued by linearized enqueues whose linearization points occurred prior to t, ordered by their enqueues' linearization order.

OptLinkedQ

The abstract head index of the queue in OptLinkedQ is defined exactly like that of OptUnlinkedQ. OptLinkedQ is our only algorithm for which the abstract state of the queue depends also on the abstract state of the tail. The abstract tail index in an execution E of OptLinkedQ is set to the index of the node enqueued by the last linearized enqueue operation. The abstract state of the queue is the sequence of items contained in the Persistent objects starting with the one enqueued by the last linearized enqueue and going through backward links until (including) the Persistent object with index bigger by 1 than the abstract head index, in reversed order; or an empty queue if the abstract tail index is not bigger than the abstract head index.

6.8 Lock-Freedom

To prove lock-freedom in the presence of crashes, we need to prove that each time a thread executes an operation on the queue, and there are no interrupting crash events since the operation's invocation, some thread completes an operation on the queue within a finite number of steps. Namely, it is sufficient to prove progress for crash-free intervals of execution. We will show that for each of the four described queue algorithms, the following holds: within n+1 loop iterations of a given running operation (assuming a crash-free long-enough interval of execution), where n is the number of threads operating on the queue, some operation succeeds to perform a linearization point. This is because an operation might fail to perform a volatile linearization point only when another operation performs a conflicting volatile linearization point, causing the original operation to retry in a new loop iteration. We argue that at a crash-free interval of execution, it is guaranteed that within a finite number of retries, some operation succeeds to reach not only a volatile linearization point, but also a survival point, thus achieving a linearization point. Hence, system-wide progress is ensured.

The same basic argument applies to all operations of all presented queues: A queue operation *op* branches backwards and starts a new loop iteration each time another operation per-

forms an obstructing volatile linearization point. If op does not succeed to pursue a volatile linearization point within *n* iterations, where *n* is the number of threads operating on the queue, then some other thread must have reached two volatile linearization points. This means it has completed the operation for which its first volatile linearization point was reached, and persisted it before returning (to satisfy durable linearizability). Thus, this operation is linearized. Yet, its linearization point might have occurred before *op*'s execution, and we need to verify that some linearization point occurs during *op*'s execution.

To prove the above argument, we start with noting that an obstructing volatile linearization point of some operation does not cause another operation to branch backwards more than once: a dequeue obstructing another dequeue has advanced the head, so the interrupted dequeue will read a new value from the queue's head in its next iteration, and an enqueue interrupted by another enqueue ensures the tail is advanced before starting a new iteration.

Next, we explain why in case of a dequeue operation, n iterations are sufficient to guarantee progress. Let the examined running op be a dequeue. It branches backwards each time another dequeue precedes it with advancing the head. If op does not complete within n iterations, some other thread must have advanced the head twice, in two different dequeue operations. This means it must have completed the first dequeue operation of the two, denoted *firstDeq*. Prior to completing *firstDeq*, the other thread has persisted the head. Thus, *firstDeq* is linearized. We still need to show that the linearization point occurs within the n inspected iterations of op and not prior to them, in order to show that n iterations of a dequeue are enough to achieve progress. *firstDeq*'s linearization point occurs in op's iteration in which *firstDeq* has failed op, because in this iteration op read the queue's head, and then failed to advance it since the obstructing *firstDeq* has advanced it in between.

For an enqueue, n iterations are not adequate to ensure progress. Let the examined running op be an enqueue. We analyze its execution since an iteration it started at moment t. op branches backwards each time another enqueue precedes it with linking a node to the tail. If a linearized enqueue fails op's first linking attempt, it is not guaranteed that the linearization point of this enqueue occurs after t. But from op's second iteration on, each enqueue that fails op and is linearized – is guaranteed to be linearized after t: it is linearized when it links its node to a previous node denoted *N*, after the tail is advanced to point to *N*, which happens after op obtains the tail in the first inspected iteration (since its obtained value must point to a preceding node, to which another enqueue operation has linked a node). Therefore, we do not look at the first n iterations of op, but rather at the n iterations starting with the second one. A similar argument to the one brought for a dequeue op applies to these iterations: If op does not complete within n+1 iterations, some other thread must have linked twice within iterations 2 to n+1, in two different enqueue operations. This means it has completed the first enqueue of the two. Prior to returning from this enqueue, it has ensured the survival point of that enqueue. Thus, this enqueue is linearized. As explained before, its linearization point occurs after t, namely, within the n+1 inspected iterations of op.

6.9 Memory Management

All queues evaluated in this work (except for OneFileQ and RedoOptQ which were adopted from [RCFC19] and [CFR20] respectively as they are with their integrated memory manager), use the same version of epoch based reclamation for memory management, called *ssmem*. This memory manager is adopted from [ZFS+19], which implements a durable extension of the mechanism presented by [DGT15] for volatile memory. *ssmem* maintains designated areas in the heap memory for node allocation. When a thread enqueues an item, it allocates a node from the next available space in these areas, or from a free list (to which dequeued nodes are inserted) if it is not empty. The memory manager keeps a persistent list of all the areas it allocated throughout the execution. During recovery, free lists are reconstructed from the unused chunks in these areas. Each thread in *ssmem* has its own allocator, operating on its separate designated areas and local free list, to avoid synchronization and reduce contention. See [ZFS+19] for more details.

6.10 Evaluation

Evaluated algorithms We compare to the durable queue in [FHMP18] as the most efficient lock-free durably linearizable queue algorithm known today. However, the queue as presented in [FHMP18] is built to satisfy more than just durable linearizability. It contains a mechanism for retrieving previously obtained results after a crash, which is not required by durable linearizability, and is not provided by other durable data structures [DDGZ18; ZFS+19]. To put all these data structures on the same level of guarantees, we remove the additional mechanism from [FHMP18], obtaining a thinner version of the original durable queue that executes faster, a version we denote durable MSQ (DurableMSQ). Comparison to the exact original queue from [FHMP18] would yield better performance for us, but would not be fair. The extra mechanism in [FHMP18] can be easily added to the versions we propose (with the corresponding additional cost).

In addition, we compare to a persistent queue implementation resulting from applying the general construction of Izraelevitz [IMS16] to MSQ. We also compare to the persistent queue version obtained by NVTraverse [FBW+20], which resembles IzraelevitzQ since the traversal phase in MSQ is empty, hence, the operations access directly the critical point, being the head or tail. The only difference between the two versions is that NVTraverseQ does not issue a fence after a flush that follows a read or CAS instruction. To complement the comparison, we compare to queues produced by wrapping a sequential queue implementation with a persistent transactional memory (PTM): OneFileQ, produced using the OneFile lock-free PTM [RCFC19], and RedoOptQ, produced using the RedoOpt PTM [CFR20].

Platform The queues were implemented in C++ and compiled using the g++ (GCC) compiler version 9.3.0 with a -O3 optimization level. We conducted our experiments on a machine running Linux (Ubuntu 18.04) equipped with 2 Intel Xeon Gold 6234 3.3GHz processors with

8 cores each. In experiments with up to 8 threads, each thread was attached to a different core of the same processor. In experiments with more than 8 threads in which the ninth and on threads were attached to the second processor, NUMA effects kick in impeding scalability and reducing performance, but the trends remain the same (OptUnlinkedQ performs best, OptLinkedQ is second best). To avoid NUMA effects, we utilize hyper-threading (SMT) on a single processor for measurements of more than 8 threads reported in Figure 6.6: we attach the $(8+i)^{th}$ thread to the second virtual core on the same physical core as the i^{th} thread.

The machine has an L1 data cache of 32KB and an L2 cache of 1MB per core, and an L3 cache of 25MB per processor. It has 1.5TB of NVRAM (Intel Optane DC Persistent Memory), organized as 128GB DIMMs (6 per processor). The machine uses the NVRAM in App-Direct Mode Interleaved in our configuration. CLWB is utilized as a flush instruction, SFENCE as a store fence and MOVNTI as a write-back to memory (non-temporal store) instruction.

Methodology In each experiment, the queue is initialized with a certain number of enqueued items, and then operations are applied to it, for five seconds unless specified otherwise. Each data point $[x, y]$ in the graphs represents the average result of 10 experiments. In each experiment, x threads performed operations concurrently. The left graphs depict the throughput, namely, number of operations applied to each queue per second by the threads altogether. The right graphs depict the throughput ratio between each queue and the baseline DurableMSQ.

We ran various workloads following prior works (see Figure 6.6): In the first workload, operations were randomly chosen to be enqueue or dequeue (50-50 uniform distribution) following [YM16; HSS07; LS04]. In the second workload, each thread ran enqueue-dequeue pairs, following measurements in [MS96; FHMP18; RCFC19; YM16; MA13; FK12; HSS07; LS04]. Next, we ran producers only (performing enqueues) on an empty queue. We also ran consumers only (performing dequeues) on a queue of size 12M following [OM20] for 1 second. At last, we ran a mixed producer-consumer workload, loosely following [OM20; HLH+13; KLP13]. Here, unlike in other workloads, the threads did not run for a preset amount of time, but rather executed a preset number of operations: one quarter of the threads performed 1M dequeues and then 1M enqueues, and the rest performed 1M enqueues and then 1M dequeues. This is intended to ensure that the queue is not drained, as enqueues are slower than dequeues. The initial queue in the presented graphs in the first, second and last workloads is of size 10. An initial size of 10K yields similar results (as we do not traverse the entire queue, but only touch the front and rear of the queue). RedoOpt is evaluated only in the first two workloads since we had problems running it on the other workloads.

Results Our first two queue designs, UnlinkedQ and LinkedQ, perform better than DurableMSQ for some workloads and worse for others. They do not gain an advantage over DurableMSQ although performing minimum fences, due to accesses to flushed cache lines. Our efficient transformations that avoid such accesses, OptUnlinkedQ and OptLinkedQ, outperform all other queues including DurableMSQ, the state-of-the-art durable queue, in nearly all exper-

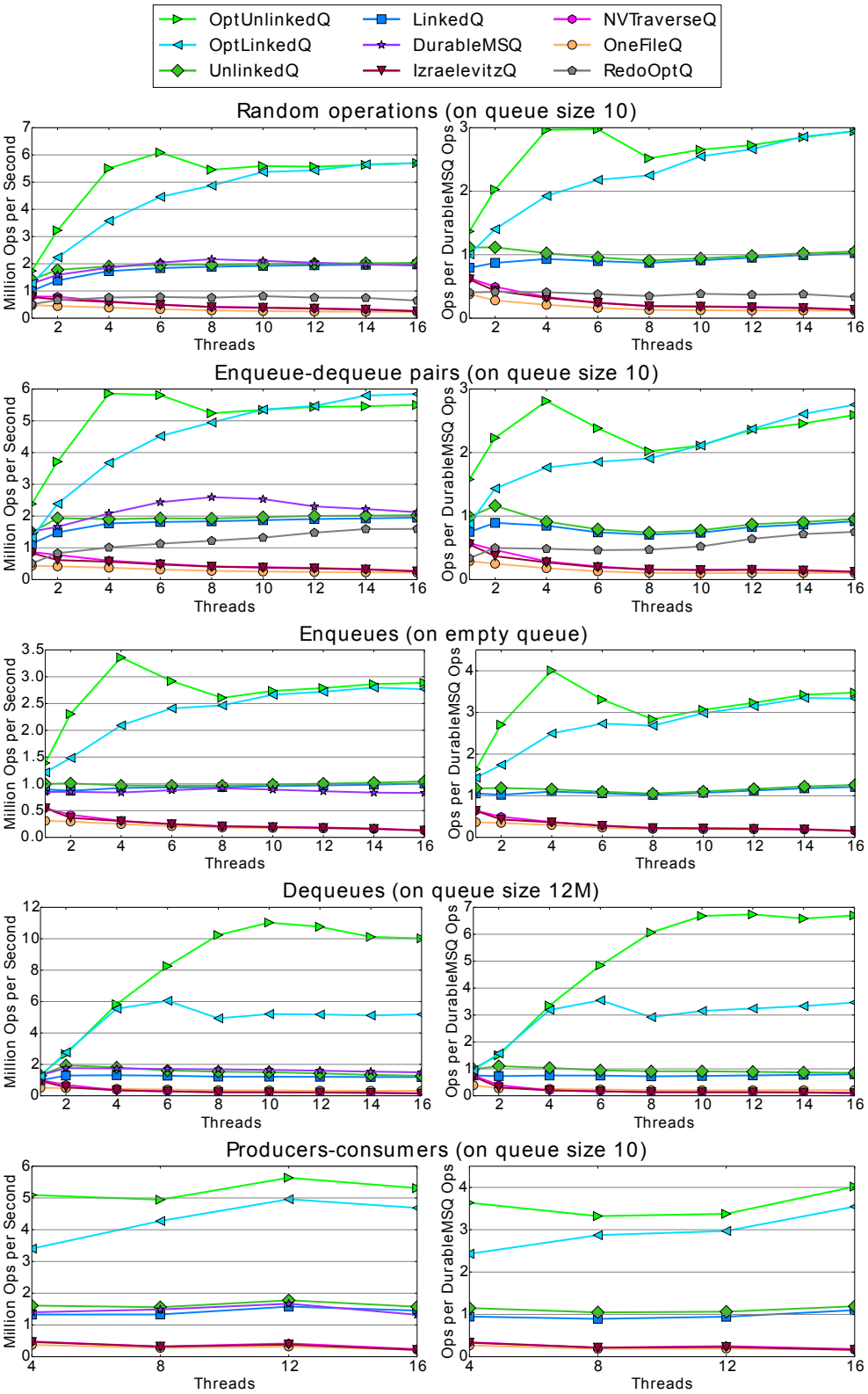


Figure 6.6: Measurement results

iments. For example, OptUnlinkedQ runs more than twice faster than DurableMSQ for nearly all workloads with more than one thread. IzraelevitzQ is substantially slower than DurableMSQ and our queues, as expected from a universal construction that places many more fences than the tailor-made queues. NVTraverseQ, which is similar to IzraelevitzQ, shows nearly identical performance. The transactional approach of OneFileQ and RedoOptQ results in reduced performance as transactions impose additional overhead over a short operation.

6.11 Conclusion

In this work we presented a new guideline for designing efficient durable algorithms suitable for the current architecture: reducing accesses to flushed memory. We demonstrated the advantage of following this guideline with durable queues. We first present novel queues that abide only to the known guideline of minimizing the fence count, meeting the theoretical lower bound on the number of fences from [CGZ18], executing only one blocking fence per operation. UnlinkedQ does not persist the links, but rather allocates the nodes on designated areas and adds an ordering mechanism, so the recovery procedure can look for valid nodes of the queue in the designated areas and order them correctly. LinkedQ uses a validity scheme on the queue nodes to inform the recovery algorithm which nodes are adequate for recovery, and adds a backward link to the queue's underlying structure to allow enqueues to persist previous enqueues efficiently. These queues do not beat state-of-the-art queues in spite of issuing fewer fences. We then amended these queues to achieve zero accesses to flushed memory while still maintaining a single blocking fence per operation. The resulted queues demonstrate a significant performance improvement on the Intel Optane NVRAM over state-of-the-art durable queues, showing that, at least in our context, the second amendment is desirable.

Chapter 7

Distributed Computations in Fully-Defective Networks

This chapter is based on the work presented at [CCGS22] and [CCGS23].

7.1 Introduction

Faults are a main hurdle in a large variety of distributed systems. Faults manifest themselves in several different manners, ranging from nodes that crash due to malfunctions to environmental disruptions that affect the communication channels connecting distant nodes. In the last few decades, research has focused on developing *fault-tolerant* algorithms, as nodes crashes and channel noise are utterly inevitable. See, e.g., recent books and surveys on fault-tolerant systems [Dub13; KK20] and algorithms [BDM93; Ray18], and references within.

In this work, we consider the case of channel noise within asynchronous distributed networks, where messages communicated between nodes are subject to corruption. When dealing with channel noise, some restrictions must be imposed on its power. Clearly, if noise can affect channels arbitrarily without any restrictions, then it could, for instance, delete all the communication and prevent any non-trivial computation over the network. Previous work either limited the *number* of channels that may suffer (arbitrary) noise [Dol82; SW90; Pel92; SAA95; HP21a; HP21b] or the *total amount* of corruptions (usually, alterations) the channels are allowed to make altogether [HS16; CGH19; GKR19; ADHS20].

Throughout this work, we consider noisy channels that may arbitrarily change the *content* of transmitted messages, but can neither delete nor inject messages. This is known in the literature as alteration noise. Yet, we do not bound the amount of noise nor the number of noisy channels in any way. That is, we ask the following question:

Can one design fault-tolerant algorithms robust to an unlimited *amount of corruption on* all *communication channels?*

On its surface, the above task seems doomed. However, we answer the question in the affirmative for the large family of 2-edge-connected networks. We further show that if the

network is not 2-edge-connected, the noise can destroy any non-trivial computation.

Towards this goal, we develop *content-oblivious* algorithms, that is, algorithms that do not rely on the *content* of communicated messages [CGH19]. Instead, the actions of a node depend on the specific links and the order in which messages are received. In particular, we devise a method that compiles *any* distributed algorithm into a content-oblivious version that computes the same task over 2-edge-connected graphs.

A folklore approach (see, e.g., in [JKL15; CGH19]) is to send a message along a certain path from u to v to signify a 0 bit, and to send a message along a different path to signify a 1 bit, where the existence of two different paths is promised by the 2-edge-connectivity property. This approach conceals many challenges. First, the edges along these two paths are also edges in paths between other nodes in the network, and so the nodes must somehow be able to associate each such "bit" with its correct origin, in order to be able to decode each original piece of information and avoid mixing up bits of different ones. Second, in order to know where to forward the message to, the nodes need to extract the sender/receiver information from these "bit" messages, yet those might be fully corrupted. Third, some guarantee needs to be obtained on the order in which different 0/1 "bits" arrive at their destination, in order for them to faithfully represent the encoded message, a caveat on which the asynchrony of the network imposes another obstacle.

Before elaborating on how we overcome all these issues and stating our main results, let us explain our setting and noise model in more detail. We abstract the network as a graph $G = (V, E)$ where every node $v \in V$ is a computing device and every edge $e \in E$ is a noisy bi-directional communication channel. Once u sends a message m over some link (u, v), the channel guarantees that after some arbitrary yet finite time, v receives some message $m' \in \{0, 1\}^+$. Note that m' may or may not equal m. In other words, the noise over the channel can corrupt the content of any transmitted m into any m', but it cannot completely delete it, nor can it inject new messages. We say that G is a *fully-defective* network if all its channels are noisy in the above manner.

7.1.1 Our Contribution and Techniques

Intuition: Content-oblivious encoding with parallel channels. Let us begin with a simple toy example that illustrates some of our techniques. Suppose u and v are directly connected by *two separate noisy channels*, which we name DATA and END. The basic idea is to communicate the information over the DATA channel by sending, "bit-by-bit", a unary encoding of the original message. In order to communicate the end of the unary encoding, a single message is sent on the END channel. Note, however, that timing is crucial: if the message sent on END is received before all the messages sent on DATA reach their destination, the receiver decodes incorrect information. To avoid this confusion, the receiver sends one message over END as an acknowledgment for each received DATA message. The sender waits until all its DATA messages are acknowledged *and only then* sends the termination message on END. Sending the terminating END message has an additional effect: it switches the roles of the nodes. If u is

the sender, then after sending the END message it takes the role of the receiver and vice versa. We call the sender at each point the *token holder*.

Main result. Since we do not wish to assume two separate channels between any two nodes, we ask whether they can be replaced with two separate *paths* between any two nodes, i.e., can we constitute reliable communication between any two neighbors in 2-edge-connected graphs?

We answer this question affirmatively and show a method that takes any asynchronous message-passing distributed algorithm π for a noiseless network G, and simulates it over the fully-defective G, given that G is 2-edge connected. By *simulating* we mean that every node has a black-box interface to π through which the node can deliver messages to π and (asynchronously) receive messages to be communicated to some neighbor. The simulation guarantees that, at any given moment, all the nodes behave similarly to some valid execution of π over the noiseless G.

Theorem 7.1 (main, informal)**.** *There exists a simulator for any asynchronous algorithm π such that executing the simulator over a 2-edge connected fully-defective network G simulates an execution of π over the noiseless network G.*

Once we establish that such a simulator even exists, a natural question is, what is the best that could be aimed for in terms of its message overhead? To avoid excessive clutter in the presentation, we delay the complete statement of our main theorem that includes its overhead, to Theorem 7.2 at the end of this section.

Warm-up: Resilient computations over a simple cycle. To describe our approach for proving Theorem 7.1, we begin with the much simpler case of *cycle graphs*. In a cycle graph, any node u is connected to only two neighbors. Our goal is to simulate u's communication with its two neighbors over a fully-defective cycle. In this special case, every two neighbors have exactly two separate paths between them: the direct link, and the rest of the cycle. The difference from the two-channel toy example illustrated above is that the paths of each two certain neighbors intersect the paths of other neighbors and we need to coordinate between the nodes so that each message reaches its correct destination and is interpreted correctly.

We address this difficulty by guaranteeing that only a single node is the sender (token holder) at any given time. All the other nodes are passive and only forward messages along the cycle. In this way, the sender can communicate with its neighbor using both paths of the cycle—one of them, say, the clockwise path, replaces the DATA channel, and the other one, the counterclockwise path, replaces the END channel.

In fact, u can use the same method to communicate with any other node on the cycle, since all the nodes see the same sequence of clockwise and counterclockwise messages. In a sense, u broadcasts information over the cycle, and all the nodes learn this information.

Next, we design a method to change the roles such that another node may become the token holder, i.e., the sender: After forwarding the message initiated by the sender in the END path, the nodes enter a *token delivery* phase. During this phase, a counterclockwise TOKEN message, initiated by the previous token holder, is forwarded along the cycle. When it reaches

a node, if the node does not have a message to send, then it forwards the token along by propagating it counterclockwise. Otherwise, it becomes the new token holder and initiates *a clockwise* message forwarded along the entire cycle which denotes the end of the token phase, so all the nodes go back to the stage of interpreting messages as DATA and END.

The above method has one significant drawback. If it takes some time for the nodes to produce a message to send, then the TOKEN message will keep circulating in the cycle, causing many superfluous transmissions. We circumvent this situation of wasteful transmissions by introducing a request mechanism: the token transfer is performed only if some node issues a request, which is done by sending a clockwise REQUEST message. The requesting node can be far away from the current token holder, thus, each node, upon receiving a REQUEST message, propagates it clockwise. We note that several nodes might issue a request at the same time at different locations on the cycle. Eventually, all nodes will have sent and received a REQUEST message, and after it reaches the current token holder, it issues the counterclockwise TOKEN message described above.

This simulator for simple cycles, in which messages are interpreted as DATA, END, TOKEN or REQUEST based only on their direction and order of transmissions, is formally given and proved in Section 7.3.

Main result: Resilient computations over 2-edge-connected graphs. To apply our approach for simple cycles to more complex graphs, we mimic it over a (not necessarily simple) cycle that goes through all the graph nodes. Such a cycle needs to be chosen carefully, because of the crucial role that the direction of messages plays in our approach. Robbins's theorem [Rob39] states that any 2-edge-connected graph G is *orientable*. That is, there exists a way to orient all edges in G so that the implied directed graph is strongly connected. This implies that there exists a cycle that goes through all nodes, possibly with multiple occurrences of some of the nodes, where all instances of any edge along the cycle bear the same orientation. We leverage the existence of such a Robbins cycle by mimicking our approach for the simple cycle over the Robbins cycle. To this end, we must first construct a Robbins cycle, as the nodes are unaware of the topology of the network. Then, we need to communicate over the Robbins cycle. Both steps are highly non-trivial and pose many challenges, as we now describe. During the first step—the construction of a Robbins cycle—we need the nodes to start communicating over partial pieces of the cycle (which are cycles by themselves), for which we need to already use the second step. For this reason, we describe the two steps in reverse order: we begin with describing the second step of communicating over a non-simple cycle, given that each node knows its previous and next neighbors along the cycle for each of its occurrences (Section 7.4). Then, we show the first step of how to construct the Robbins cycle and produce this information (Section 7.5).

Second step: Communicating over a non-simple cycle. The input of each node for this step, as will be guaranteed by our construction for the first step, is the previous and next nodes along the cycle for each of its occurrences. These inputs are consistent with some Robbins cycle, so that, in particular, each edge in the cycle has a unique orientation, and a

single orientation of the edges is considered by all nodes as the *clockwise* direction.

Mimicking our approach for a simple cycle over a Robbins cycle brings along several challenges. Consider, for instance, the network G and its induced Robbins cycle depicted in Figure 7.1. Suppose a clockwise message is received at node d along the edge (c, d). Should this message be propagated over the edge (d, e) or over the edge (d, a), or maybe over both? Note that both these options are in the clockwise direction, however, they belong to different segments of the cycle. Further, note that some messages are initiated in an asynchronous manner, e.g., the REQUEST message. Thus, when the node d receives a REQUEST message from node c, it is possible that the request originated at node a and should be propagated to node e or it originated at node e and should be propagated to node a.

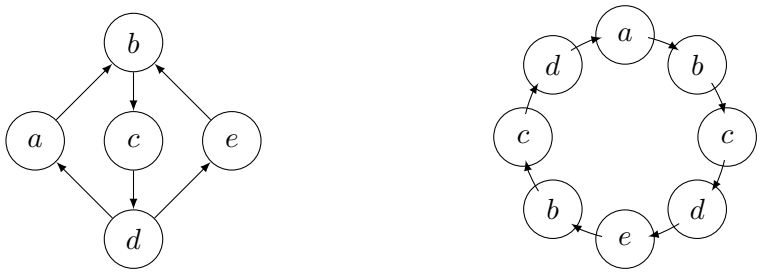


Figure 7.1: (a) A 2-edge-connected graph G with a Robbins orientation and (b) the resulting Robbins cycle with multiple occurrences per node. The arrows denote the clockwise direction of the cycle.

We cope with these issues using two separate mechanisms. The first mechanism makes sure that the TOKEN, DATA, and END messages are propagated correctly along the Robbins cycle. This mechanism consists of two main ingredients. First, we guarantee that these three message types are forwarded in a sequential manner, in the sense that the token holder issues the next message among them only after receiving the previous one from the other direction of the cycle. Second, we assure that at any given moment, each node u knows "where the token is", that is, on which segment of the cycle (i.e., between which two occurrences of u) the token resides. Since the token holder is the only node to initiate the above three message types, knowing the relative position of the token holder resolves the above and allows each node to track each message along the Robbins cycle. Indeed, each such message must first arrive from the segment in which the token holder resides and then be propagated by u to the next segment in the respective direction of the cycle. We prove that since at any given moment only one message travels through the cycle, there can be no confusion at u regarding what the message type is, which one of u's occurrences has received a message and where a message should be forwarded to.

The second mechanism we employ is for REQUEST messages. These have no pre-specified origin, and they can be initiated by any node and even by multiple nodes at the same time. The mechanism for these messages is as follows. Whenever a node receives a REQUEST message or when a node wishes to initiate one, it sends a clockwise message to *all* of its clockwise

neighbors along the Robbins cycle at the same time. Then, the node waits to receive a REQUEST message from each of its counterclockwise neighbors *and only then* it continues with executing the cycle algorithm described above. We prove that this guarantees that all nodes send and receive a request message *regardless* of their position(s) on the Robbins cycle.

First step: A content-oblivious construction of a Robbins cycle. Our Robbins cycle construction follows an *ear-decomposition* technique by Whitney [Whi32], claiming that any 2-edge-connected graph G can be decomposed into edge-disjoint parts, $G = C_0 \cup E_0 \cup E_1 \cup \cdots \cup E_k$, where C_0 is a simple cycle, and for any $0 \leq i \leq k$, E_i is an *ear*—a simple path or cycle whose endpoints belong to $C_0 \cup E_0 \cup \cdots \cup E_{i-1}$. Following Whitney's work, we iteratively decompose a 2-edge-connected graph G into some $C_0, E_0, \ldots, E_k$, part by part, and combine them into a Robbins cycle. The main obstacle we face is that our construction must be content-oblivious and cannot rely on the content of messages sent by the nodes.

The first stage of our construction is performing a DFS-like search starting from a specified *root* node. The DFS search progresses by sending a message (*a DFS-token*) sequentially, i.e., each node propagates this message to one of its unexplored adjacent edges. This DFS-token message propagates through the network until it reaches the root node again. At this stage, the path the DFS-token has taken defines a cycle C_0.

The key challenge in this stage is that the DFS-token might reach some node u twice before reaching the root. This might cause the DFS to "get stuck", e.g., if $deg(u) = 3$. We overcome this pitfall by insisting on C_0 being a simple cycle that starts and closes at the root. If some $u \neq root$ receives the DFS-token for the second time, it sends that message back on the same edge on which it was received. This has the effect of "backtracking" that edge so it is excluded from the constructed cycle. Nodes that backtrack all their adjacent edges go back to their initial state and are added to the Robbins cycle at a later step.

Once C_0 is established, the nodes on it switch to the second stage, in which they use our resilient communication approach of the above second step, in order to coordinate exploring further ears. One node on C_0 that has adjacent edges that do not belong to C_0 gets selected to initiate another DFS-like search, which again propagates in G until reaching a node on C_0, possibly different from the initiator. The path the DFS-token takes defines the ear E_0. Then, the nodes on C_0 and E_0 jointly coordinate to form a new non-simple cycle C_1 that includes all the edges in C_0 and E_0. The nodes on C_1 switch to communicate over this cycle using the above resilient communication of the second step. The nodes iterate this process, until a Robbins cycle is formed. A crucial aspect of these iterations of adding ears is that much coordination is required among the nodes for switching in a timely manner from communication on C_i to communication on C_{i+1}. The technical specification of this mechanism is given in Section 7.5.

We emphasize that the nodes do not know $|V|$, and hence they do not know when a Robbins cycle is already formed, i.e., when each node already appears on the current C_i at least once. Instead, they keep adding edges to the constructed cycle, until no node has an adjacent edge that is not in C_i, which is a state they can detect. At that point, the construction

ends.

Putting it all together. With the above two steps, our result can now be formally stated. Given any 2-edge-connected fully-defective network G and an asynchronous algorithm π designed to work on the noiseless G, we show how to compute π over the fully defective G by first constructing a Robbins cycle C on G using a resilient content-oblivious algorithm, and then simulating π over the Robbins cycle C in a resilient content-oblivious manner.

Theorem 7.2 (main)**.** *There exists a simulator for any asynchronous algorithm π, such that executing the simulator over a 2-edge connected fully-defective network G simulates an execution of π over the noiseless network G.*

The simulator has a pre-processing phase that construct a Robbins cycle C (which depends only on G) and an online phase that simulates the communication of π over C. The pre-processing step communicates $\mathsf{CC}_{\mathsf{init}} = |C|^{O(1)}$ bits. In the online phase, any message m communicated by π is simulated by communicating $\mathsf{CC}_{\mathsf{overhead}}(m) = O(|C| \cdot |m| + |C| \log |V|)$ bits.

We note that, in the worst case, $|C| = O(|V|^3)$; see Section 7.5.3 for a detailed discussion. We do not strive to optimize the polynomial overhead of our schemes, as their mere existence is the focus of this work. Nevertheless, unary encoding as explained above imposes an exponential overhead in the length of the message. In Section 7.3.3 we offer a binary encoding method that reduces the communication complexity to the polynomial terms stated above.

Impossibility result. We complement the above result and show that if G is not 2-edge-connected, then there is no way to conduct non-trivial computations over a fully-defective G. To this end, we prove the following impossibility for *two-party computation* over a fully-defective channel. The impossibility for a non 2-edge-connected G follows since it contains a bridge, and we can reduce the two sides of the bridge to the two-party case.

Theorem 7.3. *Fix a non-constant function $f(x, y)$. No two-party deterministic algorithm that gives output or terminates can compute f over a fully-defective channel.*

The theorem requires the nodes to either terminate or irrevocably give an output. Note that the above theorem differs from the famous *two generals coordinated-attack* impossibility [Gra78], since our noise model does not allow deleting messages. See Section 7.6 for complete details.

7.1.2 Related Work

There are two common ways to deal with channel corruptions. One is by adding redundancy, i.e., coding the information, an approach that is known in the literature as *Interactive Coding*. The other is by diverting the communication so it would not pass through corrupted edges, which are known as *Byzantine edges*.

We review some related work in these areas, but we stress that neither approach can be used in fully-defective networks: Interactive coding must assume some bound on the errors,

either per channel or globally, while solutions for networks with Byzantine edges must assume a bound on the number of noisy channels.

Interactive coding was initiated by the seminal work of Schulman [Sch92; Sch93; RS94], see [Gel17] for a recent survey on this field. In this setting, communication channels either suffer from stochastic noise [RS94; GMS14; BEGH17; GK19; ABE+19] or from some bounded amount of adversarial noise. E.g., if limiting the overhead of the coding scheme to be linear, [GMS14; HS16; JKL15; LV15; GKR19] develop schemes resilient to up to a fraction $O(1/|E|)$ of the total communication. Without any restriction on the overhead, schemes can cope with noise up to a fraction $O(1/|V|)$ of the total communication, and such a fraction is shown to be maximal [JKL15]—otherwise, the adversarial noise could completely corrupt all the outgoing communication of the node that communicates the least. The above works assume synchronous networks. Censor-Hillel, Gelles, and Haeupler [CGH19] developed the first coding scheme for *asynchronous* networks that suffer from up to a fraction $O(1/|V|)$ of adversarial noise. Communication with an unbounded (yet, *finite*) amount of noise was examined in [DMSY15; ADHS18; GI20] for the two-party case and in [ADHS20] for the multiparty case. In a work by Efrmenko, Haramaty, and Kalai [EHK20], the noise model is similar to the one we consider here in the sense that it can corrupt *the content* of messages but not their existence. However, the amount of bit-corruptions in [EHK20] (measured as the edit distance between sent and received messages) is bounded to a constant fraction out of the entire communication. Furthermore, their work considers only two parties.

Networks with Byzantine edges do not restrict the amount of noise per link, and even allow insertion/deletion errors, but allow only a bounded number of links to be noisy. In asynchronous settings, Fisher, Lynch, and Paterson [FLP85] exclude the existence of consensus algorithms when a single node may crash, or equivalently, when all the links connected to some single node may crash. In synchronous networks, certain tasks are also impossible with arbitrary link failures [Gra78; SWK09]. On the other hand, Santoro and Widmayer [SW90] considered distributed function evaluation when (a large number of) links suffer either corruptions, insertions, deletions, or their combination. In a sense, the synchrony guarantee allows simpler solutions, e.g., encoding information via the time in which messages are sent. Pelc [Pel92] shows that if the number of Byzantine links is bounded by f, robust communication is achievable only over graphs whose edge-connectivity is more than $2f$. This is also implied by the work of Dolev [Dol82]. Additional works [PT86; GLR95; SCY98; Das98; Bie03] consider the case of mixed node and link failures.

Recent work by Hitron and Parter [HP21a; HP21b] gives a compiler that turns any algorithm in the noise-free setting into an algorithm that works correctly even if the adversary controls f edges in a $(2f+1)$-edge-connected network. The above is for the synchronous Congest setting. Their approach is to construct a family of low-congestion cycle-covers (see also [PY19a; PY19b]), which are structures in which for every edge (u, v), there are at least $f+1$ cycles that contain no adversarial edges. We stress that low-congestion cycle-covers do not seem to be helpful for our setting: Even if we were promised only *two* cycles that share a *single* edge, it is not clear how to communicate over them in a way that distinguishes one

from the other.

7.2 Preliminaries

Notations. We use $a\|b$ or $a \cdot b$ for the concatenation of a and b. For a positive integer $k \in \mathbb{N}$ and a string b, we let $b^k = b \cdot b \cdot \cdots \cdot b$ denote b concatenated to itself for k times; $b^0 = \epsilon$ is the empty string. For a string b and an integer $0 \leq i \leq |b| - 1$, we let b_i denote the i-th bit of b, i.e., $b = b_0 b_1 \cdots b_{|b|-1}$.

Networks and protocols. A protocol π over an undirected network $G = (V, E)$ with $n = |V|$ nodes is an asynchronous event-driven distributed algorithm, in which nodes conduct some computation by sending messages to their neighbors in G (for simplicity, we assume only deterministic algorithms in this work). Upon the reception of a message, π instructs the recipient node what message(s) to send next, as a function of the node's input and all the messages it has received so far. Specifically, each node v begins with a private input x_v (which may be empty), and knowledge of the IDs of its neighboring nodes, $N(v) = \{u \mid (u, v) \in E\}$ (we can remove this assumption, see Remark 6). According to the input to v, π generates messages to send to zero or more of v's neighbors (possibly different messages to different neighbors). Afterwards, the protocol behaves in an event-driven manner, i.e., nodes act only upon receiving messages: whenever a node v receives a message, it performs some computation and produces messages designated to zero or more of its neighbors. We impose no assumption on the computation time of π except that it is finite. We additionally assume a preselection of one designated node (which will function as a root node in our Robbins cycle construction), and assume that every node knows whether it is the designated node.

Communicating a message over some link of G takes arbitrary positive finite time. Channels are *not* assumed to be FIFO. Incoming messages are kept in an incoming buffer until processed by the node.

The *protocol's transcript* τ of a given execution, is the sequence of messages sent and received during the execution. Each item in τ indicates the message sent or received, the sending or receiving node and the link on which the message was communicated. Events that happen in different nodes at the same time are assumed to be ordered in some arbitrary order. The *local transcript* τ_v of a node v, is the ordered sequence of messages sent and received by v. Note that τ_v can be derived from τ as the sub-sequence in which v is the sending or receiving node.

We say that π *gives an output* if every node eventually writes an output to its write-only output register. This action is irrevocable. If needed, the node may remain active and send and receive messages after giving an output; that is, we do not require termination, but our result also applies to protocols that terminate. We say that the protocol has reached *quiescence* at some time, if no message is still in transit and from that time on, no new messages are sent over the network.

Fully-defective networks and noise-resilient simulations. We work in networks with

noisy channels exposed to alteration noise, which can corrupt the *content* of any message communicated over any channel. That is, once a message $m \in \{0,1\}^+$ is sent over some link, the received message may be any $m' \in \{0,1\}^+$. However, the noise *cannot* completely delete a message nor can it inject a message on a link in which no m was sent. We stress that, except for inserting and deleting messages, the noise has no restrictions at all. In particular, it can apply to all channels and corrupt all messages in a given execution. We call networks that suffer noise as specified above *fully-defective* networks. Equivalently, one can think about such a network as one in which nodes communicate only by means of sending pulses to their neighbors, which could be the case, for instance, when the nodes have very basic communication hardware.

A *noise-resilient simulator* designed for a noiseless network $G = (V, E)$ is a protocol $\hat{\pi}$ which is given as an input an asynchronous black-box interface to some π. When $\hat{\pi}$ is executed on a fully-defective network G, it produces for each node $v \in V$ a string $\hat{\tau}_v$, such that there exists some execution of π over the noiseless network G that generates a transcript τ, for which $\tau_v = \hat{\tau}_v$ for each node v. We allow a simulator to perform some pre-processing before simulating π. We define $\mathsf{CC}_{\mathsf{init}}$ to be the communication complexity in bits of the simulator during the pre-processing, and $\mathsf{CC}_{\mathsf{overhead}}(m)$ to be the communication complexity for simulating the delivery of a message m. Note that CC accounts only for the length of *sent* messages, even if later their content is corrupted by the noise.

Distributed representation of cycles. A (directed) cycle can be represented in a distributed network in two manners: *locally* and *globally*. A local representation of some cycle C means that every node on C knows its two neighbors on the cycle along with their respective direction, clockwise or counterclockwise, usually held in the local variables $next$ and $prev$, respectively. In case C is not a simple cycle, then every node knows its clockwise and counterclockwise neighbors for each of its occurrences on C. This information is consistent across all nodes in the sense that an outside observer who follows the neighbors and directions of each node would see a consistent directed cycle.

A global representation of a directed cycle means that every node $v \in C$ holds the string $C = (v_1, v_2, \ldots)$ of the IDs of the nodes on C in their clockwise order.

7.3 Simulating Computations over a Fully-Defective Simple Cycle

As discussed in Section 7.1, we can establish a resilient connection between two nodes connected by two separate links, sending *content-less* messages between them, which we will call *pulses* throughout this chapter. Our goal is to implement this idea for any two nodes in a 2-edge-connected graph, since in such a graph any two nodes are connected by two separate *paths*. As a stepping stone, in this section we consider the special case of *simple cycles*.

Theorem 7.4 (A simulator for a simple cycle)**.** *There exists a noise-resilient simulator for any asynchronous protocol π and any fully-defective simple cycle G in which each node knows its*

clockwise and counterclockwise neighbors. The simulator features $\mathsf{CC}_{\mathsf{init}} = 0$ *and* $\mathsf{CC}_{\mathsf{overhead}}(m) = O(|V| \cdot |m| + |V| \log |V|)$ *pulses.*

Let G be a simple cycle on $V = \{v_i\}_{0 \leq i \leq n-1}$ with $E = \{(v_i, v_{i+1})\}_{0 \leq i \leq n-1}$, where indices are taken mod n. The main idea is to imitate the two-channel idea described in Section 7.1 above over the cycle. That is, suppose v_i wishes to send a message to its neighbor v_{i+1}. We can think of the link (v_i, v_{i+1}) as the DATA channel, and on the *path* $v_i, v_{i-1}, v_{i-2}, \ldots, v_{i+1}$ as the END channel. For this to work, all the nodes beside v_i and v_{i+1} need to simply forward each pulse they receive along the same direction. However, the above description supports only a single fixed sender and a single fixed receiver. Thus, we need a method that allows different nodes to become the sender. For this we use a token mechanism, where only a single node holds the token at any given time.

Our simulator can be split into two separate phases per message transfer: the first one is the *token phase* which handles transferring the token between the nodes, and the second one is the *data phase* that handles communication between the current token-holder and the rest of the nodes.

The token phase works as follows. At the starting point, there exists only a single token holder. During the token phase, pulses carry one out of two possible meanings: either they are a REQUEST pulse or a TOKEN pulse. The meaning of a pulse is dictated by the direction in which the pulse progresses along the cycle: REQUEST is a clockwise pulse while TOKEN is a counterclockwise pulse. A node that wishes to obtain the token issues a REQUEST pulse. Every node that receives such a REQUEST pulse, propagates it in the same direction, unless it has already sent a REQUEST pulse previously in this phase, so eventually every node sends and receives *a single* REQUEST pulse.

Upon receiving a REQUEST pulse, the current (single) token holder releases the token by sending a counterclockwise TOKEN pulse. This pulse propagates along the cycle until it reaches one of the nodes which requested the token. A node that receives the TOKEN pulse and wishes to become a token holder does not propagate the TOKEN pulse but instead sets itself as the new token holder. Then, the new token holder switches to its data phase and begins sending clockwise pulses, which are interpreted as DATA pulses. The first of these pulses propagates throughout the entire cycle and informs all the other nodes that the token phase has completed. This first pulse cannot be confused with a REQUEST pulse since we guarantee that every node sends and receives exactly a single REQUEST pulse in each token phase. In other words, the second clockwise pulse received during a token phase must be a DATA pulse, which triggers its recipient to switch to its data phase.

In the data phase, the token holder delivers its message via a unary coding. That is, it sends a number of clockwise DATA pulses that equals the length of the unary encoding of the information. Each node other than the token holder forwards each received DATA pulse clockwise, so these pulses propagate along the cycle until they reach the token holder back from the other side of the cycle. Then, the token holder sends a single counterclockwise END pulse that signals the end of the message and the end of the data phase. Note that once the

token holder receives the END pulse from the other direction, all nodes know that the data phase is over, and are back in the token phase. Note also that due to the asynchrony, nodes that already moved to the next token phase might send a REQUEST pulse before the END pulse arrives at the token holder. Our design promises that these REQUEST pulses are not confused with pulses of the current data phase: END pulses are sent in the other direction, and as for DATA pulses—the token holder does not proceed to sending a REQUEST pulse before it receives the END pulse of the data phase, so REQUEST pulses of the new token phase can only reach nodes that have already received the END pulse for this phase and therefore do not interpret them as additional DATA pulses.

A *phase* is a local concept, in the sense that each node runs a specific data or token phase in any given time, and different nodes might be in different phases in a certain time. We denote each token phase and its subsequent data phase an *epoch*. An epoch is a local concept too, viewed by each node according to the phase it is currently running. Different nodes might be in different epochs in a certain time: some nodes might already send a REQUEST pulse in the new epoch while others have still not received an END pulse for the previous epoch.

7.3.1 Formal Description

We now formally describe our simulator over fully-defective simple cycles, where each node is given the identities of its clockwise and counterclockwise neighbors. Our simulator receives as an input an asynchronous protocol π for noiseless communication channels. Messages to be sent are generated by π, and any message received by a node in our simulator is delivered and processed by π. Our simulator thus treats π as an asynchronous black box that interfaces with the simulator by sending and receiving messages, internally at each node. We stress that π's actions take finite arbitrary time unknown to and independent of the simulator algorithm.

Our simulator appears in Algorithms 1(a) and 1(b). All nodes begin executing the token phase (Algorithm 1(a)). Each node u has an internal $isTokenHolder_u$ variable that indicates whether it is the token holder. Moreover, each node u keeps a queue Q_u of messages generated by π, which should be broadcast over the cycle. Messages in Q_u are of the form (m, u, v), where m is a message that π instructs u to send to v. At the onset, $isTokenHolder_u$ is True for a single node, and each Q_u is empty. When π gives an output, the respective node gives the same output in the simulator but keeps executing the communication algorithm over the cycle. If in a certain time all the queues $\{Q_u\}$ are empty and remain empty, then the simulator stops sending messages and reaches quiescence.

The simulator is content-oblivious, and as such it communicates by sending pulses (contentless messages). Note that in our algorithms we write next to each pulse its meaning (DATA, END, REQUEST, TOKEN), however, this is only for the analysis; the nodes assign this meaning according to their current state and the clockwise/counterclockwise direction of the pulse, and not by the pulse content, which is ignored.

Algorithm 1(a) A simulator for simple cycles: token phase (node u)

Init: A single node has $isTokenHolder = \mathsf{True}$. Node u holds a (possibly empty) input x_u for π.

Handling messages sent by π: During the execution of the algorithm, node u enqueues to Q_u any new message π asks u to send, in the form (message, source, destination). The actions of π occur in parallel to the execution of this algorithm.

1: **wait until** Q_u is not empty or a clockwise REQUEST pulse is received
2: send a clockwise REQUEST pulse
3: **if** no clockwise REQUEST pulse was received **then** wait until receiving a clockwise REQUEST pulse **end if**
4: **if** $isTokenHolder_u$ **then**
5: $\quad isTokenHolder_u \leftarrow \mathsf{False}$
6: $\quad$ send a counterclockwise TOKEN pulse
7: **end if**
8: **wait until** receiving a pulse
9: **if** the pulse is a counterclockwise TOKEN pulse **then** ▷ Else, the pulse is a clockwise DATA pulse
10: $\quad$ **if** Q_u is not empty **then**
11: $\quad\quad isTokenHolder_u \leftarrow \mathsf{True}$
12: $\quad$ **else**
13: $\quad\quad$ forward the counterclockwise TOKEN pulse
14: $\quad\quad$ **go to** Line 8
15: $\quad$ **end if**
16: **end if**
17: continue with Algorithm 1(b)

7.3.2 Analysis

Let us set some notation for the analysis of Algorithm 1. Let '1' indicate a pulse sent clockwise, and let '0' indicate a pulse sent counterclockwise. Recall that an *epoch* is the execution of consecutive token and data phases. We say that a node has completed its k-th epoch once it has executed Line 31 for the k-th time. Let T_k be the k-th node to have set its $isTokenHolder_u$ to True in Line 11, whereas T_0 is the node whose $isTokenHolder_u$ variable is initialized to True. (We will show that T_k sends, in its k-th epoch, the k-th simulated message in the system.) Let s_k, for $k \geq 1$, be the time in which T_k sets its $isTokenHolder \leftarrow \mathsf{True}$ ($s_k = \infty$ if T_k is undefined). Finally, let t_k be the time in which T_k completes its k-th epoch ($t_k = \infty$ if T_k is undefined or never ends the k-th epoch). We let $s_0 = t_0 = 0$. Let $[u \curvearrowright v]$ denote the clockwise path from u to v along the cycle including both ends, and similarly let $[u \curvearrowleft v]$ denote the counterclockwise path from u to v. In the special case of identical endpoints, $[u \curvearrowright u]$ denotes the path through the whole cycle. To exclude an endpoint, we use a round bracket in place of a square bracket, e.g. $[u \curvearrowright v)$ denotes the clockwise path excluding v; the

Algorithm 1(b) A simulator for simple cycles: data phase (node u)

18: **if** $isTokenHolder_u$ **then**
19: dequeue a message from Q_u, denote it by (m, u, v) and let 1^d be its unary encoding
20: send d clockwise DATA pulses
21: **wait until** receiving d clockwise DATA pulses
22: send a counterclockwise END pulse
23: **wait until** a counterclockwise END pulse is received
24: **else**
25: forward any received clockwise DATA pulse **until** receiving a counterclockwise END pulse
▷ Including the DATA pulse received during the preceding token phase
26: let $count$ be the number of received clockwise DATA pulses
27: decode 1^{count} as the unary encoding of (m', u', v')
28: **if** $u = v'$ **then** deliver m' to π (as if received from u') **end if**
29: forward the counterclockwise END pulse
30: **end if**
31: continue with Algorithm 1(a)

path can be empty, i.e., $(u \rightsquigarrow v)$ for u, v neighbors.

Our analysis is based on the following technical lemma, which provides us with three important properties satisfied by Algorithm 1 in every epoch: (1) **progress,** which says that as long as there is a message to send, the next epoch will eventually start and complete; (2) **single token holder,** which says that at most a single node holds the token at any moment (there is no such node during the time in which the token is being passed), and T_k is the only one to hold it during the data phase of the k-th epoch; and (3) **global consistency,** which says that in any given epoch k, exactly one message is being communicated—sent by T_k and received by all other nodes, and the pattern of pulses every node sends has a distinct structure. We now formalize these ideas as follows.

Lemma 7.3.1. *Consider an execution of Algorithm 1 and consider any $k \geq 1$, for which $t_{k-1} < \infty$. If from time t_{k-1} and forward, all queues $\{Q_v\}_{v \in V}$ are always empty, then $t_k = \infty$. Otherwise, the following hold:*

*(1) **Progress:** All nodes eventually complete their k-th epoch. In particular, $t_k < \infty$, and at time t_k, all nodes have already processed the END pulse (of epoch k) but have not yet passed Line 8 in epoch $k+1$ (they are either waiting in Lines 1 or 3 for a REQUEST pulse, or waiting in Line 8 for either a DATA or a TOKEN pulse).*

*(2) **Single token holder:** It holds that $t_{k-1} < s_k < t_k$. At each moment in (t_{k-1}, t_k), there is at most a single node for which $isTokenHolder = \mathsf{True}$. More specifically, within this time frame, the token is passed as follows: the node T_{k-1} releases the token at some time in $[t_{k-1}, s_k)$ and the node T_k is the next node that gains the token at time s_k. The node T_k (solely) holds the token in $[s_k, t_k]$.*

(3) ***Global consistency:*** *There exist integers $d_1, \ldots, d_k > 0$ and for any $u \in V$ there are $b_1^u, \ldots, b_k^u \in \{0,1\}$, such that when the node u completes its k-th epoch, its sent transcript (the overall pulses sent so far by u) is $P_{u,k} \triangleq 10^{b_1^u}1^{d_1}0 \cdot 10^{b_2^u}1^{d_2}0 \ldots 10^{b_k^u}1^{d_k}0$.*

In addition, the message each node decodes and processes (Lines 26–28) in its k-epoch is the unary decoding of 1^{d_k}, which is the message sent by T_k (Line 19) in its k-th epoch.

Proof We prove the statement by induction on the epoch number k. We start with proving the base case, $k = 1$. The proof for the general case is very similar. The analysis follows the progress of the protocol and shows that each pulse sent with a certain meaning (i.e., DATA, END, TOKEN, REQUEST) is correctly interpreted by its recipient.

Base Case, $k = 1$. Note that $t_0 = 0$, hence, $t_{k-1} < \infty$. All the nodes begin by executing Algorithm 1(a), with a single node T_0 having $isTokenHolder = \mathsf{True}$. While all nodes have empty queues Q_v, they all wait in Line 1 and thus, if the queues remain empty indefinitely, we have $t_1 = \infty$.

Otherwise, at some time there is at least one node u that enqueues to Q_u a message to be simulated. Each such u sends a REQUEST pulse in Line 2 and waits to receive a REQUEST pulse in Line 3, unless it has already received such a pulse in Line 1. As there is at least one such node, at least one REQUEST pulse is sent. The rest of the nodes first wait to receive a REQUEST pulse and then forward it. It follows that all nodes eventually receive and send a single REQUEST pulse. Let us denote by $\tilde{P}_u$ the partial transcript of a node u at the "current" time (which evolves with the proof), then $\forall u \in V$, we have $\tilde{P}_u = 1$ after sending the REQUEST. To prove Property (3), we keep track of the partial transcript $\tilde{P}_u$, recording the pulses sent by each node.

After sending and receiving a REQUEST pulse, any node $u \neq T_0$ waits to receive another pulse (Line 8). The node T_0 sets $isTokenHolder_{T_0} = \mathsf{False}$, sends a counterclockwise TOKEN pulse (Line 6) and then waits for another pulse like all other nodes. The TOKEN pulse triggered by T_0 propagates counterclockwise until it reaches a node v with a non-empty Q_v, which must exist. The node v subsequently sets $isTokenHolder_v$ to True (Line 11). Thus, by the above definitions, we get that $T_1 = v$ and s_1 is the time when v executes Line 11. Note that T_1 might get the TOKEN pulse before getting a REQUEST pulse, in which case it delays its actions until a REQUEST pulse is received. This has no effect on the proof.

In case $T_1 \neq T_0$, at time s_1, all the nodes on $[T_0 \rightharpoonup T_1)$ have sent a TOKEN pulse and are now waiting for a DATA pulse (Line 8) that would switch them to their data phase of epoch $k = 1$. The nodes on $(T_0 \rightharpoondown T_1)$ could be in two possible stages: either they are still waiting for a REQUEST pulse (Lines 1 or 3) as described above, or they are waiting in Line 8. Hence at time s_1, every node $u \in [T_0 \rightharpoonup T_1)$ has $\tilde{P}_u = 10$, while every node $u \in (T_0 \rightharpoondown T_1]$ has $\tilde{P}_u = \epsilon$ if it has not yet sent a REQUEST pulse, or $\tilde{P}_u = 1$ otherwise. Eventually, perhaps at a different time per node, each node u thus reaches the partial transcript $\tilde{P}_u = 10^{b_1^u}$ with $b_1^u \in \{0,1\}$ being the indicator of whether u has sent a TOKEN pulse (namely, whether $u \in [T_0 \rightharpoonup T_1)$).

In the special case where $T_1 = T_0$, at time s_1, the token has just reached back at T_1; all nodes have sent a TOKEN pulse, and all nodes but T_1 are now waiting for a DATA pulse (Line 8)

that would switch them to their data phase of epoch $k = 1$. Hence at time s_1, every node u has the partial transcript $\tilde{P}_u = 10^{b_1^u}$ with $b_1^u = 1$ indicating that u has sent a TOKEN pulse.

When the node T_1 switches to the data phase (Algorithm 1(b)), its queue Q_{T_1} is non-empty and so it sends $d \geq 1$ clockwise DATA pulses (Line 20). We define $d_1 = d$. These DATA pulses propagate clockwise through all nodes, after the first received DATA pulse in each node but T_1 triggers it to switch to its data phase, after it has previously received a REQUEST pulse. Note that each such node must have received a REQUEST pulse before it receives the first DATA pulse. This is because its counterclockwise neighbor, who sends the DATA pulse, moves to the data phase only after it has sent a clockwise REQUEST pulse.

Once a node $u \neq T_1$ is in its data phase of epoch $k = 1$, it records each received DATA pulse. The node propagates the pulse clockwise and eventually the pulse arrives back at T_1. Thus, since T_1 sends d_1 DATA pulses, after propagating them, each node u has $\tilde{P}_u = 10^{b_1^u} 1^{d_1}$. Once the d_1 clockwise pulses reach back at T_1, and only then, it issues a counterclockwise END pulse (Line 22). T_1 does not generate nor does it propagate any additional pulses before receiving the propagated END from the other side of the cycle. This implies that any $u \neq T_1$ receives exactly d_1 clockwise DATA pulses followed by a counterclockwise END pulse. Upon receiving the END pulse, u processes the message 1^{d_1} (Lines 26–28) and forwards the END pulse (Line 29). It then completes its k-th data phase and its k-th epoch, with $\tilde{P}_u = 10^{b_1^u} 1^{d_1} 0$. The node T_1 also has $\tilde{P}_{T_1} = 10^{b_1^{T_1}} 1^{d_1} 0$ when it receives the END pulse and switches to the next token phase, at time t_1. At that time, all the other nodes have already processed the END pulse. This proves the first part of Property (1).

Next, we need to prove that at time t_1, none of the nodes has passed Line 8. In order for a node to pass Line 8, it must be the case that *after* the node has switched to the token phase, it has received a REQUEST pulse followed by one additional pulse (in any direction). We argue this cannot happen. Indeed, at the time where some node v receives the END pulse and switches to its second token phase, only the nodes in $(T_1 \rightharpoonup v]$ have received the END pulse and only these nodes have switched to their (second) token phase. In their token phase, they may or may not have sent a clockwise REQUEST pulse by this time. Thus, only nodes in $[T_1 \rightharpoonup v)$ might have *received* a REQUEST pulse. However, it is impossible that they received an additional pulse by time t_1, as we next show. Each node in $(T_1 \rightharpoonup v)$ that has received a REQUEST pulse is waiting to receive another pulse (Line 8) and is not generating any pulse. The node T_1, if receiving a REQUEST pulse, does not process it and does not send a REQUEST pulse before receiving an END pulse in Line 23. It also never sends a pulse in the counterclockwise direction before receiving its END pulse back. Furthermore, each node in $(T_1 \rightharpoondown v)$ (for $v \neq T_1$) is still executing Line 25, so it only forwards pulses and never generates pulses. Finally, v has just started its token phase and is waiting to receive a REQUEST. We conclude that no additional pulse (beyond the REQUEST pulse, if sent) can be received by the nodes in $(T_1 \rightharpoonup v)$. This holds for any v at the time it transitions to its second token phase. It thus holds for all nodes at time t_1, when the END pulse eventually reaches back at T_1.

Next we prove Property (2) based on the above description of the first epoch. At the onset (at time t_0), T_0 is the only node with $isTokenHolder_{T_0} = \mathsf{True}$. As mentioned above, T_0 sets

$isTokenHolder_{T_0} = \mathsf{False}$ and sends a TOKEN pulse during its token phase. The propagated TOKEN pulse is the one that triggers T_1 to set $isTokenHolder_{T_1} = \mathsf{True}$ later, at time s_1. Thus, it is clear that $s_1 > t_0 = 0$, and that T_0 releases the token before s_1 and T_1 becomes a token holder at s_1. Later, at time t_1, the node T_1 completes the first epoch, hence, $t_1 > s_1$. The node T_1 does not set $isTokenHolder = \mathsf{False}$ during the time frame $[s_1, t_1]$, and it remains to show that it is the only token holder throughout this time frame.

It is clear that no node in $(T_0 \rightsquigarrow T_1)$ has set itself as a token holder as otherwise, that node would have been the node we indicate as T_1. After time s_1, no more TOKEN pulses are sent in the first token phase. Further, T_1 sends a clockwise DATA pulse that transitions all other nodes into their data phase. This implies that no node besides T_1 can execute Line 11 and set $isTokenHolder = \mathsf{True}$ in this token phase. As for the second token phase, each node that reaches it before t_1 does not pass Line 8 before time t_1, as we have shown above, thus in particular, it does not receive a TOKEN pulse and does not reach Line 11.

Finally, we prove Property (3), that is, that all nodes reach a global consistency regarding the sent message of the first epoch. This follows from the above analysis: As we argued, at the time some node u completes its first epoch, it holds that $\tilde{P}_u = 10^{b_1^u} 1^{d_1} 0$. The part 1^{d_1} corresponds to the d_1 DATA pulses initiated by T_1, which form the encoding of the message communicated in this epoch. This completes the proof of Property (3).

Induction Step. To complete the proof, we need to prove the induction step. Most of the above proof holds as is for $k > 1$, if we replace s_1, t_1, T_1 with s_k, t_k, T_k, etc.

Fix $k > 1$ with $t_{k-1} < \infty$ (otherwise, the lemma holds vacuously). We use the induction hypothesis on epoch $k - 1$. We are allowed to do so since $t_{k-1} < \infty$, which implies that at or after time t_{k-2} there is at least one non-empty Q_v and the three properties of the lemma apply to epoch $k - 1$. The differences between proving the base case and the step are as follows:

In the case where all nodes have an empty Q_v, it is immediate in the base case that no node ever passes Line 1; we prove the same happens here. However, all the induction hypothesis gives us is that at t_{k-1} all nodes are waiting either in Line 1 or 3, or 8. Clearly, nodes cannot be in Line 3 since their queue is empty. We now prove they cannot be in Line 8 as well.

Assume towards a contradiction that v is the first to pass Line 1 in its k-th epoch. Let $\tilde{t}$ be the time when v receives the END pulse of its epoch number $k - 1$. After time $\tilde{t}$, the node v completes its epoch and transitions to its k-th token phase. Since Q_v is empty, v gets to Line 1 and awaits there for a REQUEST pulse. Because v eventually reaches Line 8, it must have received a clockwise pulse from its neighbor u, which caused v to pass Line 1.

For $v \neq T_{k-1}$, recall that at time $\tilde{t}$, the nodes $[T_{k-1} \rightsquigarrow v)$ are still in their data phase after receiving d_{k-1} DATA pulses. Recall also that they do not generate new pulses but only propagate pulses, and they do not propagate any additional clockwise pulses because T_{k-1} does not generate any clockwise pulses until the counterclockwise END reaches it. The above-mentioned neighbor u belongs to $[T_{k-1} \rightsquigarrow v)$, hence, it does not propagate further clockwise pulses from time $\tilde{t}$ until u gets the END pulse. In case $v = T_{k-1}$, u gets the END pulse before time $\tilde{t}$.

In any case, after u gets the END pulse, u transitions to its k-th token phase and reaches Line 1. Therefore, if u did send a REQUEST pulse that causes v to pass Line 1, then u would have also passed Line 1 prior to sending this REQUEST pulse, in contradiction to our assumption that v is the first node to pass Line 1.

If some node has a non-empty Q_v, in the base case we had that all nodes begin the token phase at the same time t_0, and then send a REQUEST pulse if their queue is non empty or if they receive a REQUEST pulse. When considering the induction step at time t_{k-1}, some nodes may have already started their k-th epoch, and have already sent a REQUEST pulse before time t_{k-1}, as given by Property (1) of the induction hypothesis. The behavior from this point on remains the same as described above for the base case.

For proving the global consistency property in the induction step, let $P_{u,k-1}$ be the transcript of u at the end of its epoch $k-1$. By Property (3) of the induction hypothesis, we know that there exist $d_1, \ldots, d_{k-1}$ and $b_1^u, \ldots, b_{k-1}^u$ for any $u \in V$ such that $P_{u,k-1} \triangleq 10^{b_1^u} 1^{d_1} 0 \cdot 10^{b_2^u} 1^{d_2} 0 \ldots 10^{b_{k-1}^u} 1^{d_{k-1}} 0$ for any $u \in V$. Furthermore, the above analysis shows that there exist an integer $d_k > 0$ and an indicator $b_k^u \in \{0,1\}$ per u, such that the pulses sent by node u during its k-th epoch can be described by $10^{b_k^u} 1^{d_k} 0$, where 1^{d_k} signifies the DATA pulses sent by T_k, which encodes the communicated message of this epoch. This gives Property (3). ■

Next, we show that Lemma 7.3.1 implies the correctness of the simulator (Theorem 7.5). We then analyze its overhead (Lemma 7.3.2). Finally, we discuss in Section 7.3.3 ways to improve the obtained complexity (Lemma 7.3.4). Together, these three prove Theorem 7.4.

Theorem 7.5. *Let $G = (V, E)$ be a cycle. For any asynchronous protocol π, let $\hat{\pi}$ be the protocol defined by Algorithm 1 with the input π. Then, executing $\hat{\pi}$ on the fully-defective G simulates an execution of π on the noiseless network G.*

Proof Let $\mathcal{E}_{\hat{\pi}}$ be an execution of $\hat{\pi}$ over the fully-defective cycle G. We derive a transcript τ from $\mathcal{E}_{\hat{\pi}}$, and claim that τ corresponds to a valid transcript of some execution of π in the noiseless network G, which we denote $\mathcal{E}_{\pi}$. The reader should distinguish between the simulated π which is the black-box interface used as an input of Algorithm 1, and the protocol π that generates the execution $\mathcal{E}_{\pi}$ on the noiseless network G. In order to avoid confusion, we will use the term *simulated* π to denote the former and refer to $\mathcal{E}_{\pi}$ when discussing the latter.

Let us specify the structure of the transcript τ. We can think about it as an ordered sequence of events $\tau = \tau_1 \tau_2 \cdots$, where τ_i is either the event that some node u sent a message m to v, i.e., $\tau_i = (\langle \mathsf{send} \rangle, u, v, m)$ or the event that some node u received a message m from v, i.e., $\tau_i = (\langle \mathsf{receive} \rangle, u, v, m)$.

To derive τ from $\mathcal{E}_{\hat{\pi}}$, we follow the execution of $\hat{\pi}$ as the time evolves. We add a *send* event every time the simulated π instructs node u to send a new message m. Specifically, when node u enqueues $M = (m, u, v)$ to Q_u, we add the event $(\langle \mathsf{send} \rangle, u, v, m)$ to τ. Additionally, every time some node v delivers the message $M = (m, u, v)$ to the simulated π (Line 28), we

add the event $(\langle \mathsf{receive} \rangle, v, u, m)$ to τ. Recall that π generates messages to u sequentially and τ maintains this order; events that happen at the same time in different nodes are ordered arbitrarily in τ.

Given $\mathcal{E}_{\hat{\pi}}$ and its derived τ, we prove that there exists an execution $\mathcal{E}_\pi$ of π on the noiseless network G that produces these exact same events in the same order, i.e., such that τ is exactly the transcript of $\mathcal{E}_\pi$.

The execution $\mathcal{E}_\pi$ is obtained by executing π over G with the following scheduler that imitates the behavior of $\mathcal{E}_{\hat{\pi}}$. Every time a node sends a message in $\mathcal{E}_\pi$, the message is delayed at the channel and delivered only at the time the respective message is received in $\mathcal{E}_{\hat{\pi}}$. That is, our scheduler "follows" the execution of $\mathcal{E}_{\hat{\pi}}$, and delays each message until the time its corresponding message is delivered in $\mathcal{E}_{\hat{\pi}}$. Specifically, whenever a *receive* event is registered in τ (in $\mathcal{E}_{\hat{\pi}}$), we deliver the corresponding message in $\mathcal{E}_\pi$. The scheduler also controls the execution time of all nodes, which enables it to control the timing of the send events π initiates in $\mathcal{E}_\pi$ so they correspond to the same order of send events in $\mathcal{E}_{\hat{\pi}}$. We now show that the above defines a valid scheduler, and that the resulting $\mathcal{E}_\pi$ has the transcript τ.

Define $\mathsf{time}(j)$ to be the time in $\mathcal{E}_{\hat{\pi}}$ when the event τ_j is registered (note, for multiple events that occur at the same time, we let $\mathsf{time}(j)$ refer only to the events up to τ_j). We prove the following statement by induction on j: (1) The scheduler is valid: whenever instructed to deliver a message m, this m was issued to the channel and hasn't been delivered yet. (2) The transcript τ derived from $\mathcal{E}_{\hat{\pi}}$ is a prefix of the transcript of $\mathcal{E}_\pi$.

For the base case, $\mathsf{time}(0)$, the transcript τ is empty. It is clear that the scheduler is (vacuously) valid, and that τ is a prefix of the transcript of $\mathcal{E}_\pi$.

We proceed with the induction step. Assume that the induction statement holds at $\mathsf{time}(j-1)$, that is, at this point in time, the events $\tau_1 \cdots \tau_{j-1}$ are a prefix of the events in $\mathcal{E}_\pi$, and all the actions of the scheduler so far are valid. Now consider the next event recorded to τ. There are two options here, either it is a send event or a receive event.

In the first case, let $\tau_j = (\langle \mathsf{send} \rangle, u, v, m)$. Consider $\mathcal{E}_\pi$ right after the event τ_{j-1}, i.e., at $\mathsf{time}(j-1)$. Every node u in $\mathcal{E}_\pi$ has exactly the same state as the simulated u in the simulated π, which follows from the induction hypothesis. Therefore, if the simulated π instructs u to send the message m to v (which triggers τ_j in $\mathcal{E}_{\hat{\pi}}$), the same (eventually) happens at node u in $\mathcal{E}_\pi$. The scheduler delays all other nodes until the same message is sent in $\mathcal{E}_\pi$, and the claim thus holds after event τ_j, i.e., at $\mathsf{time}(j)$ as well.

The other case is when the j-th event is a receive event, say, $\tau_j = (\langle \mathsf{receive} \rangle, u, v, m)$. Consider the node u that executes Line 28 which corresponds to this event. By Property (3) of Lemma 7.3.1, this message is sent by the message sender of that epoch, v. Denote this epoch by k. This means that at the beginning of epoch k the message m appeared in Q_v and was dequeued by v at the beginning of the data phase of epoch k; note that dequeued messages are never enqueued back to Q_v. This means that at some point in time before $\mathsf{time}(j)$, node v enqueued this message to Q_v and it was never dequeued before the k-th epoch; let τ_i with $i < j$ be the corresponding event of enqueuing m to Q_v. Now consider $\mathcal{E}_\pi$. By the induction hypothesis we know that up till event τ_{j-1} at $\mathsf{time}(j-1)$, the transcript τ describes the

execution $\mathcal{E}_\pi$, thus, the message m was sent at $\mathsf{time}(i)$ and is currently being delayed by the channel (since the first and only delivery of m in $\mathcal{E}_{\hat{\pi}}$ occurs at $\mathsf{time}(j)$). The scheduler then instructs the channel to deliver this message, which is a valid action as this message was already issued to the channel and never delivered before. This completes the inductive proof.

The above proves that at any point in time the execution $\mathcal{E}_{\hat{\pi}}$ over the fully-defective G simulates a prefix of a valid execution of π over the noiseless G. It remains to show liveness, namely, that the prefix keeps growing. This follows from Properties (1) and (3) of Lemma 7.3.1: the simulation makes progress as long as some Q_v is non-empty or eventually becomes non-empty. Progress means that all nodes begin and complete their next epoch. In each epoch one message (from some Q_v) is being delivered to its destination. If the simulated π of some node gives an output, the same node will give the same output in $\mathcal{E}_{\hat{\pi}}$. If we consider the point in time where all nodes have given output, then all these outputs are valid since τ at that time is a prefix of the execution $\mathcal{E}_\pi$, which also gives the same outputs, by definition.

The only situation where the simulation could reach quiescence is when all the queues Q_v are empty and remain empty indefinitely. But τ up to that time, as argued above, is a transcript of some $\mathcal{E}_\pi$ on the noiseless G, where no messages are currently delayed by any channel, and no new messages are going to be sent since the nodes in $\mathcal{E}_\pi$ are at the same state as in the simulated π. Thus, $\mathcal{E}_\pi$ has reached quiescence as well. ■

Let us point out a couple of additional remarks about our simulation.

Remark. **FIFO:** The scheduler derived from our simulator maintains FIFO: Consider an execution $\mathcal{E}_{\hat{\pi}}$ of the simulator $\hat{\pi}$. If multiple messages from u to v exist in the simulation, they are enqueued and communicated by their order. These enqueues translate in $\mathcal{E}_\pi$ to messages sent over the same link. However, the scheduler for $\mathcal{E}_\pi$ will deliver them according to their order in $\mathcal{E}_{\hat{\pi}}$'s transcript, which is their order in Q_u, that maintains a FIFO property. This strengthens our result, that is, the simulation works correctly both with or without FIFO assumptions for the simulated protocol.

Remark. **No starvation:** Our proof shows that as long as some u has a message to send, then *some message* will be sent during the next epoch. Since the TOKEN pulse travels counterclockwise sequentially in the cycle, there can be at most $n-1$ epochs until u becomes the token holder. Thus, our simulator actually satisfies the stronger notion of *no starvation*.

Remark. **Broadcast:** We note that by design, our simulator offers an additional *broadcast* operation. That is, a node can send a message whose destination is all other nodes. To provide this functionality, we utilize the fact that every message arrives at all nodes, regardless of its original destination. To broadcast a message m, a node simply fixes its destination to be $*$. Each node that decodes a message delivers it to π if its destination is either that node (as before) or $*$. We will use this feature in our Robbins cycle construction in Section 7.5.

Lemma 7.3.2. *The overhead of simulating a single message m in Algorithm 1 is* $\mathsf{CC}_{\mathsf{overhead}}(m) = |V|^{O(1)} \cdot 2^{|m|}$.

Proof Let $n = |V|$ be the length of the cycle. Suppose the message $M = (m, u, v)$ is dequeued in some epoch and is being communicated (i.e., m is being communicated by the simulated π over the link (u, v)). We can write $|M| = |m| + O(\log n)$. Communicating M over the cycle results in the following pulses sent by *each* node during this epoch (Property (3) of Lemma 7.3.1): a single REQUEST pulse, at most a single TOKEN pulses, $2^{|M|}$ DATA pulses and a single END pulse. Since there are n nodes, where each node sends at most $3 + 2^{|M|}$ pulses, we conclude that $\mathsf{CC}_{\mathsf{overhead}}(m) = O(n \cdot 2^{|m|+O(\log n)}) = n^{O(1)} \cdot 2^{|m|}$. ■

7.3.3 Reducing the Communication via Binary Encoding

Encoding each message via a unary encoding leads to a pulse overhead that is exponential in the message size: $\mathsf{CC}_{\mathsf{overhead}}(m) = \text{poly}(n, 2^{|m|})$, with $n = |V|$. We now show how to send messages over a simple cycle via a binary encoding of the message. This binary encoding leads to a much improved communication complexity of $\mathsf{CC}_{\mathsf{overhead}}(m) = O(n \cdot |m| + n \log n)$.

Let $M = (m, u, v) \in \{0, 1\}^*$ be the message that the token holder u wishes to send. The idea is to encode the bits of M so that a clockwise pulse denotes the bit 1, and a counterclockwise pulse denotes the bit 0; we denote these as DATA(1) and DATA(0), respectively. Since the order of the bits is important, the token holder sends the next bit only after receiving the previous bit from the other direction of the cycle. (As an optimization, all the pulses of consecutive same-bit sequences may be sent concurrently, and then the token holder should wait to receive all the pulses of the same direction before sending pulses in the other direction. For clarity of the presentation, we do not delve into the details.) However, now that counterclockwise pulses signify a 0 data bit, the challenge is that we need a different way to indicate the end of transmitting M, that is, we need a way to encode an END pulse.

We overcome this challenge by having the nodes agree on a fixed parameter L. In order to communicate that M's transmission has completed (replacing the END pulse), the token holder sends $L \geq 2$ consecutive counterclockwise pulses. In addition, the bitstring M is padded with a 1 after every $L-1$ consecutive 0s (when read from its first symbol and onward). An additional trailing 1 is sent after $pad(M)$ and guarantees that, even if M has ended with a 0 or a sequence of 0s, then $pad(M) \cdot 1 \cdot 0^L$ has L consecutive 0s only at its suffix. Furthermore, we add a preceding 1 before $pad(M)$: Recall that the token holder must initiate the sending protocol with a clockwise DATA pulse, as otherwise, the sender's first counterclockwise pulse might be mistaken for a TOKEN pulse in those nodes that have not yet forwarded a TOKEN pulse and are still in the token phase. To summarize, in order to communicate the message M, the token holder communicates pulses according to the encoded message $Z = 1 \cdot pad(M) \cdot 1 \cdot 0^L$.

The revised data phase algorithm is given in Algorithm 2.

We argue that replacing Algorithm 1(b) with Algorithm 2 does not change the premise of Theorem 7.5. For the rest of this section, we change Line 17 in Algorithm 1(a) to say "continue with Algorithm 2".

We show that an execution of Algorithm 1(a) along with Algorithm 2 satisfies a *Global consistency* property similar to the one of Lemma 7.3.1. One can easily verify that the *Progress*

Algorithm 2 Data phase: Broadcasting a message, binary version (node u)

An integral parameter $L \geq 2$ is agreed upon all nodes.

1: **if** $isTokenHolder_u$ **then**
2: dequeue a message from Q_u and denote it as $M = (m, u, v)$
3: let $pad(M)$ be the string obtained from M by inserting a 1 after every $L-1$ consecutive 0s of M
4: $Z \leftarrow 1 \cdot pad(M) \cdot 1 \cdot 0^L$
5: **for** $j = 0$ to $|Z| - 1$ **do**
6: **if** $Z_j = 1$ **then**
7: send a clockwise DATA(1) pulse
8: wait until a clockwise pulse is received
9: **else**
10: send a counterclockwise DATA(0) pulse
11: wait until a counterclockwise pulse is received
12: **end if**
13: **end for**
14: **else**
15: **repeat**
16: forward every incoming pulse along the cycle, according to its original direction
17: record each clockwise pulse as a 1 and each counterclockwise pulse as a 0
18: **until** L consecutive 0s have been recorded
19: let Z be the recorded string. Parse $Z = 1 \cdot P \cdot 1 \cdot 0^L$
20: let $M' \leftarrow pad^{-1}(P)$ be the string obtained by removing any 1 that appears after a sequence of $L-1$ consecutive 0s. Parse $M' = (m', u', v')$
21: **if** $u = v'$ **then** deliver m' to π (as if received from u') **end if**
22: **end if**
23: continue with Algorithm 1(a)

property and the *Single token holder* property hold as well, with the same proof as before.

Lemma 7.3.3. *Consider the following modification of the* ***Global consistency*** *property:*

> *There exist strings $z_1, \ldots, z_k$, where $z_i = 1m_i10^L$ for some $m_i \in \{0,1\}^+$, and for any $u \in V$ there are $b_1^u, \ldots, b_k^u \in \{0,1\}$, such that when the node u completes its k-th epoch, its sent transcript (the overall pulses sent so far by u) is $P_{u,k} \triangleq 10^{b_1^u} z_1 \cdot 10^{b_2^u} z_2 \cdot \cdots \cdot 10^{b_k^u} z_k$.*
>
> *In addition, the message each node decodes and processes (Line 20 in Algorithm 2) in epoch k is exactly the message $pad^{-1}(m_k)$ sent by T_k.*

Then the statement of Lemma 7.3.1 holds for the simulator given by Algorithms 1(a) and 2.

In order to avoid excessive repetition, we sketch below only the differences from the proof of Lemma 7.3.1 that stem from replacing Algorithm 1(b) with Algorithm 2.

Proof Recall from the proof of Lemma 7.3.1, that T_k gains the token during its k-th token phase (at time s_k) since its Q_{T_k} is non-empty and a TOKEN pulse arrives from its counterclockwise neighbor. The node T_k then switches to its data phase (Algorithm 2).

The node T_k dequeues a message M from its queue Q_{T_k} and sets $z_k = 1 \cdot pad(M) \cdot 10^L$ in Line 4. Thus, its first pulse is a clockwise DATA(1) pulse, which causes every other node u to switch to its data phase and execute the code with $isTokenHolder_u = \mathsf{False}$, similarly to the case in the proof of Lemma 7.3.1, upon receiving the first DATA.

Note that T_k transmits bits sequentially and proceeds to the next bit only after the previous pulse is received from its other side of the cycle. That is, once the first DATA(1) pulse arrives back at T_k, it continues to communicating $pad(M) \cdot 10^L$, bit after bit.

The padding $pad(M)$ and the trailing 1 following it guarantee that there exists only a single substring of L consecutive 0s in z_k, which resides at the suffix of z_k. It follows that all other nodes receive the string z_k: they record the message communicated by T_k bit by bit, until they see L consecutive 0s. This sequence appears only at the suffix of z_k and signifies its termination. We can thus deduce that the message Z recorded by each node has the structure $Z = 1 \cdot P \cdot 10^L$ so each node continues to extracting the part P (whose length is unknown beforehand) and decodes $M' = pad^{-1}(P)$ to obtain the correct message $M' = M$ communicated by T_k in Line 20. If the node is the recipient of M it delivers it to its simulated π (Line 21). Each node then completes its k-th epoch, and transitions to its token phase $k+1$ (Algorithm 1(a)).

Since every received pulse is forwarded along the same direction it was received, during the k-th epoch each node u transmits exactly the sequence of pulses described by $10^{b_k^u} z_k$, and thus its overall sent transcript is $P_{u,k} = P_{u,k-1} \cdot 10^{b_k^u} z_k$, which has the correct structure using the induction hypothesis. The rest of the proof follows the one of Lemma 7.3.1 as is. ■

Lemma 7.3.4. *The overhead of simulating a single message m by Algorithm 1(a) and Algorithm 2 over the simple cycle G is $\mathsf{CC}_{\mathsf{overhead}}(m) = O(n \cdot |m| + n \log n)$.*

Proof Let $n = |V|$ be the length of the cycle G. Suppose the message $M = (m, u, v)$ is dequeued in some epoch and being communicated (i.e., m is being communicated by the simulated π over the link (u, v)). Communicating M over the cycle results in the following pulses sent by *each* node during this epoch (Property (3) of Lemma 7.3.3): a single REQUEST pulse, at most a single TOKEN pulses, and at most $2 + L + (1 + \frac{1}{L-1})|M|$ DATA pulses (a preceding and trailing 1s, L trailing 0s, and $|pad(M)| \leq (1 + \frac{1}{L-1})|M|$ "content" pulses). Since $L \geq 2$ is a constant, each of the n nodes sends $O(|M|) = O(|m| + \log n)$ pulses. We conclude that $\mathsf{CC}_{\mathsf{overhead}}(m) = n \cdot O(|M|) = O(n \cdot |m| + n \log n)$. ■

7.4 Simulating Computations over Fully-Defective 2-Edge Connected Networks

In this section we show how to perform resilient computations over any 2-edge-connected fully-defective network, given a Robbins cycle.

Theorem 7.6 (A simulator for a Robbins cycle). *Let C be a Robbins cycle (over G) and let each node know its clockwise and counterclockwise neighbors for each of its occurrences on C. There exists a noise-resilient simulator over the fully-defective G for any asynchronous protocol π. The simulator features $\mathsf{CC}_{\mathsf{init}} = 0$ and $\mathsf{CC}_{\mathsf{overhead}}(m) = O(|C| \cdot |m| + |C| \log |V|)$ pulses.*

Let G be a 2-edge-connected graph, and assume the nodes are given a Robbins cycle C, namely, a directed cycle that passes through each vertex at least once, and that does not use any edge in both of its directions. (We stress that this assumption is later removed by showing how to construct the Robbins cycle from scratch, in Section 7.5.) As a node u may appear in C more than once, we denote by k_u the number of its occurrences on C. The initial knowledge of each node u about C is the value of k_u and its clockwise and counterclockwise neighbors along the cycle. That is, every node u knows the nodes $prev_{u,i}$ and $next_{u,i}$ for every $0 \leq i \leq k_u - 1$, such that the nodes along the cycle C correspond to the $prev$ and $next$ variables of all nodes in a consistent manner. We refer to the sequence of nodes between two successive occurrences of u on C (including the ending occurrence of u) as a *segment* $(u \to \cdots \to u]$.

The high level approach for the algorithm is built upon Algorithm 1 of the simple cycle, with pulses forwarded across the Robbins cycle C. By way of mimicking the protocol for the simple cycle, u views each of its occurrences along C as a different node along the cycle. Accordingly, when a node u is the token holder, it has exactly one occurrence on C which is associated with holding the token, and when we refer to a token holder in this section, we refer to that precise occurrence. When any node u forwards a pulse in some direction, it forwards it to the node along C that follows its occurrence that received the pulse. There are several challenges in this generalization.

Challenge 1: Edge repetition along C. Perhaps the main challenge is for u to keep track of its occurrences and distinguish between them: it could be that multiple occurrences of u have the same incoming edge. Still, this node needs to be able to associate each pulse it receives with its appropriate segment, even when pulses that belong to different segments arrive from the same neighbor.

For instance, consider the node d in Figure 7.1, and suppose it has just started its data phase and received a clockwise DATA pulse from node c. This data pulse could have originated at node e and should be forwarded to node a, or it could have originated at node a and should be forwarded to node e.

To avoid this type of confusion, each node u tracks throughout the execution in which of its segments the token is located. Specifically, the node u maintains the invariant that $prev_{u,i}$ and $next_{u,i}$ reflect the previous and next nodes of its occurrence number i, for $0 \leq i \leq k_u - 1$, in the specific rotation of the cycle that **starts** with the token segment considered by u, i.e., the token always resides within segment 0, (locally) for all nodes. To achieve this, node u applies a local rotation function upon receiving information about the token holder, namely, upon receiving a TOKEN pulse which we show that can be traced correctly to a specific segment.

Challenge 2: Distinguishing between different DATA pulses. Another challenge that

arises is how to distinguish between different DATA pulses. Recall that in the simulator for the simple cycle, the d DATA pulses are forwarded concurrently, in the sense that the token holder issues all d DATA pulses and only then waits to receive them. However, once an edge appears more than once in C, its endpoint u needs to tell apart the case in which it receives two different DATA pulses on that edge from the case in which it receives the same DATA pulse on that edge but from different segments. This is crucial because d is not known in advance (and in fact the value of d is the exact piece of information that needs to be learned).

We overcome this challenge by making sure that the DATA pulses get forwarded in a sequential manner as follows. The node-occurrence that is the token holder does not issue all d DATA pulses, but rather waits to receive DATA pulse number ℓ from its counterclockwise neighbor before issuing DATA pulse $\ell + 1$, for $1 \leq \ell \leq d - 1$. A node u that receives the DATA pulse for the i-th time since the last reception of a counterclockwise END pulse, forwards it to $next_{u,i-1}$ (where the index is taken mod k_u).

Challenge 3: REQUEST pulses have no guaranteed structure. While our approach for overcoming Challenges 1 and 2 allows the nodes to have consistent rotations of the cycle and the token segments for streamlining the DATA, END, and TOKEN pulses, it is insufficient for handling REQUEST pulses. The reason for this is that each of the other three types of pulses traverses the cycle sequentially (or partially traverses in case of a TOKEN pulse), but REQUEST pulses could be initiated by different nodes, so that a node that receives a REQUEST pulse does not have any particular promise about its origin and hence cannot tell which neighbor to forward this pulse to.

We remedy this uncertainty by having each node disseminate REQUEST pulses to all of its clockwise neighbors, regardless of their origin (which is not known to the node). We show that in the case of REQUEST pulses, this coarse action satisfies the conditions that are needed in order for the simulator to work correctly, despite its somewhat more aggressive and unstructured nature.

7.4.1 Formal Description

The main idea of the simulator, as mentioned above, is to let each node mimic Algorithm 1 while simulating each one of its occurrences on C as if it were a separate node on a simple cycle. Nevertheless, some actions are performed by the node and apply for all its occurrences. We expand on this shortly.

In particular, each node u has the internal variables $isTokenHolder_u$ and Q_u, for holding the token and queuing its simulated messages. These will serve all its occurrences. Recall that a segment $(u \to \cdots \to u]$ is a sub-path of the cycle C between two consecutive occurrences of u. Each node u holds the variables $prev_{u,i}$ and $next_{u,i}$ that reflect the previous and next nodes of its occurrence number i, for $0 \leq i \leq k_u - 1$, see Figure 7.2.

Further, each node u tracks throughout the execution in which of its segments the token is located and calls this its *token segment* (segment 0). The node u applies a local rotation function RotateEdges() upon receiving information about the token holder, namely, upon

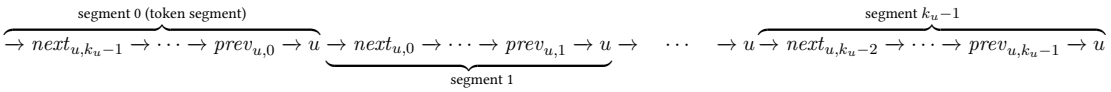


Figure 7.2: The segments of the rotation of C that starts with the token segment, as seen by a specific node u. The token resides in one of the node-occurrences or links of the token segment.

receiving a TOKEN pulse, which maintains this invariant. The procedure RotateEdges() is formally defined as follows.

RotateEdges() for node u: Update each $prev_{u,i}$ to the previous value of $prev_{u,i-1}$ and each $next_{u,i}$ to the previous value of $next_{u,i-1}$, where indices are taken mod k_u.

The pseudo-code of our simulator appears in Algorithms 3(a) and 3(b) below. We are now ready to prove its correctness and analyze its communication complexity.

7.4.2 Analysis

Similar to the analysis in Section 7.3.2, we begin by proving the technical Lemma 7.4.1 that specifies the behavior of the simulation and replaces Lemma 7.3.1. This technical lemma is then used to argue the correctness of our simulation over a Robbins cycle (Theorem 7.7). We prove the complexity of our simulation in Lemmas 7.4.2 and 7.4.3. Together, these prove Theorem 7.6.

For the analysis, we use the same notations as in Algorithm 1, up to the following modification. Since we have to be careful and distinguish between the different occurrences of a node on C, we let T_k denote the following: Consider the k-th node to have set its $isTokenHolder_u$ to True in Line 12. T_k is the *node-occurrence* of this node that has received a TOKEN and subsequently set $isTokenHolder_u$ to True.

Lemma 7.4.1. *Consider an execution of Algorithm 3 and consider any $k \geq 1$, for which $t_{k-1} < \infty$. If from time t_{k-1} and forward, all queues $\{Q_v\}_{v \in V}$ are always empty, then $t_k = \infty$. Otherwise, the following hold:*

(1) ***Progress:*** *All nodes eventually complete their k-th epoch. In particular, $t_k < \infty$, and at time t_k, all nodes have already processed the END pulse (of epoch k) but have not yet passed Line 8 in epoch $k+1$ (they are either waiting in Lines 1 or 3 for a REQUEST pulse, or waiting in Line 8 for either a DATA or a TOKEN pulse).*

(2) ***Single Token Holder:*** *It holds that $t_{k-1} < s_k < t_k$. At each moment in (t_{k-1}, t_k), there is at most a single node u for which $isTokenHolder_u =$ True and u associates this with a single occurrence on C. More specifically, within this time frame, the token is passed as follows: the node-occurrence T_{k-1} releases the token at some time in (t_{k-1}, s_k) and the node-occurrence T_k is the next node that gains the token at time s_k. The node-occurrence T_k (solely) holds the token in $[s_k, t_k]$.*

(3) ***Global consistency:*** *There exist integers $d_1, \ldots, d_k > 0$ and for any $u \in V$ there are $b_1^u, \ldots, b_k^u \in \{0,1\}$, such that when the node u completes its k-th epoch, the sent transcript by each of its occurrences is $P_{u,k} \triangleq 10^{b_1^u} 1^{d_1} 0 \cdot 10^{b_2^u} 1^{d_2} 0 \ldots 10^{b_k^u} 1^{d_k} 0$.*

In addition, the message each node decodes and processes (Line 39) at its k-epoch is the unary decoding of 1^{d_k}, which is the message sent by T_k (Line 20).

Proof In essence, we wish to follow the line of proof of Lemma 7.3.1. The high-level observation is that in the general case, every *occurrence* of a node on C behaves as in the case of the

Algorithm 3(a) A simulator for fully-defective networks given a Robbins cycle: token phase (node u)

Init: A single node has $isTokenHolder =$ True, associated with one specific occurrence. Node u holds variables $next$ and $prev$ for each one of its occurrences, so that these variables (globally) form a Robbins cycle. The first segment (of u on the cycle) contains the node-occurrence associated with the token. Node u holds a (possibly empty) input x_u for π.

Handling messages sent by π: During the execution of the algorithm, node u enqueues to Q_u any new message π asks u to send, in the form (message, source, destination). The actions of π occur in parallel to the execution of this algorithm.

1: **wait until** Q_u is not empty or a clockwise REQUEST pulse is received from some $prev_{u,i}$
2: send a REQUEST pulse to $next_{u,i}$ for all $0 \leq i \leq k_u - 1$
3: **wait until** a REQUEST pulse is received on each $prev_{u,i}$ for all $0 \leq i \leq k_u - 1$
▷ Including REQUEST pulses received in Line 1, if any
4: **if** $isTokenHolder_u$ **then**
5: $isTokenHolder_u \leftarrow$ False
6: send a counterclockwise TOKEN pulse to $prev_{u,0}$
7: **end if**

8: **wait until** receiving a pulse
▷ Or process any *second* pulse received in Line 3 from $prev_{u,i}$ for some $0 \leq i \leq k_u - 1$
9: **if** the pulse is a counterclockwise TOKEN pulse **then** ▷ Else, the pulse is a clockwise DATA pulse
10: RotateEdges()
11: **if** Q_u is not empty **then**
12: $isTokenHolder_u \leftarrow$ True
13: **else**
14: forward the counterclockwise TOKEN pulse to $prev_{u,0}$
15: **go to** Line 8
16: **end if**
17: **end if**
18: continue with Algorithm 3(b)

Algorithm 3(b) A simulator for fully-defective networks given a Robbins cycle: data phase (node u)

19: **if** $isTokenHolder_u$ **then**
20: dequeue a message from Q_u, denote it by (m, u, v) and let 1^d be its unary encoding
21: **for** d times **do**
22: **for** i from 0 to $k_u - 1$ **do**
23: send a clockwise DATA pulse to $next_{u,i}$
24: **wait until** receiving a clockwise DATA pulse from $prev_{u,(i+1) \mod k_u}$
25: **end for**
26: **end for**
27: **for** i from $k_u - 1$ to 0 **do**
28: send a counterclockwise END pulse to $prev_{u,(i+1) \mod k_u}$
29: **wait until** a counterclockwise END pulse is received from $next_{u,i}$
30: **end for**
31: **else**
32: **repeat**
33: **for** i from 0 to $k_u - 1$ **do**
34: **wait until** receiving a clockwise DATA pulse from $prev_{u,i}$
▷ Including the DATA pulse received in the preceding token phase, if exists
35: forward the clockwise DATA pulse to $next_{u,i}$
36: **end for**
37: **until** receiving a counterclockwise END pulse
38: let $count$ be the total number of clockwise DATA pulses received by u divided by k_u
39: decode 1^{count} as the unary encoding of (m', u', v')
40: **if** $u = v'$ **then** deliver m' to π (as if received from u') **end if**
41: **for** i from $k_u - 1$ to 0 **do**
42: **wait until** a counterclockwise END pulse is received from $next_{u,i}$
▷ for $i = k_u - 1$ this is already received in Line 37
43: forward the counterclockwise END pulse to $prev_{u,i}$
44: **end for**
45: **end if**
46: continue with Algorithm 3(a)

simple cycle, rather than every node behaving that way. There are a few subtle exceptions to this, which do not harm the proof but are rather essential for allowing it to go through. We elaborate as follows.

Token Phase. For the token phase, if a node u has a non-empty queue Q_u, then it reaches Line 2 and sends a REQUEST pulse to *each* of its clockwise neighbors, $next_{u,i}$ for all $0 \leq i \leq k_u - 1$, and waits in Line 3 to receive a REQUEST pulse from *each* of its counterclockwise neighbors, $prev_{u,i}$ for all $0 \leq i \leq k_u - 1$. This is equivalent to saying that every occurrence of u on C sends a single clockwise REQUEST pulse and waits to receive a single

counterclockwise REQUEST pulse. Thus, Lines 1–3 are equivalent to Lines 1–3 of Algorithm 1 for each occurrence of u.

Similarly, any node u that receives a REQUEST pulse from $prev_{u,j}$ for some $0 \leq j \leq k_u - 1$ in Line 1, forwards it to each of its neighbors $next_{u,i}$ for all $0 \leq i \leq k_u - 1$, and waits in Line 3 to receive a REQUEST pulse from each of its neighbors $prev_{u,i}$ for all $0 \leq i \leq k_u - 1$ for which $i \neq j$. For occurrence j of u, this is equivalent to Lines 1–3 of Algorithm 1. For other occurrences of u, this is slightly different, as they first forward the REQUEST pulse and only then wait to receive it. However, this still satisfies that if some REQUEST pulse is sent in an epoch, then each occurrence of every node sends and receives exactly one REQUEST pulse in that epoch, which is all that is needed for the proof of Lemma 7.3.1. Notice that in Line 3 a node may receive a second clockwise pulse from $prev_{u,i}$ for some single $0 \leq i \leq k_u - 1$, in which case this is a DATA pulse that is processed in Line 8.

Consider now the node u which has $isTokenHolder_u$ set to True. In Lines 4-6, this node sends a counterclockwise TOKEN pulse to $prev_{u,0}$. This corresponds to having only occurrence 0 of u on C send a TOKEN pulse, which is the same as Lines 4–6 in Algorithm 1 and indeed the proof of Lemma 7.3.1 needs that only a single TOKEN pulse traverses the cycle.

It remains to show that each occurrence of a node u on C that receives a TOKEN pulse forwards it to its counterclockwise neighbor in C if the queue Q_u is empty, or sets $isTokenHolder_u$ to True otherwise, and that there is exactly one occurrence of one node among those with a non-empty queue which receives a TOKEN pulse. For this, we need to show that each node correctly keeps track of its token segment. We rely on the local RotateEdges() procedure by showing that the following invariant holds in any time throughout the execution: Let $v*$ be the node-occurrence of v that is associated with v having $isTokenHolder_v$ set to True, or the node-occurrence that most recently received a TOKEN pulse if no such v exists. Then for every node u, the occurrence $v*$ is located in segment 0 of u on C (i.e., $v*$ is located in $[next_{u,k_u-1}, \ldots, prev_{u,0}, u]$). This invariant is assumed to hold at the onset of the execution. Since the invariant holds, once a TOKEN pulse reaches a node u in Line 9, it must reach it from $next_{u,k_u-1}$. Then, u invokes RotateEdges() in Line 10 before setting $isTokenHolder_u$ to True in Line 12 or forwarding the TOKEN pulse in Line 14, depending on whether its queue Q_u is empty. In either case, the invocation of RotateEdges() guarantees the invariant is maintained.

Now, consider the case where a node v sets its $isTokenHolder_v$ to True. For the node-occurrence $v*$, Lines 8-17 correspond to Lines 8-16 in Algorithm 1, that is, $v*$ receives a TOKEN pulse and switches to its data phase. For the other occurrences of v this is slightly different, as the node v along with all its occurrences, switches to its data phase once $v*$ obtains the token and node v executes Line 18 after completing Line 12. This is fine, since all the occurrences at this point have sent and received a REQUEST pulse, and can switch to the data phase. Indeed, some of v's node-occurrences might have already forwarded a TOKEN pulse before and remained in the token phase (e.g., if Q_v was empty once the token had reached them); these occurrences switch to the data phase "late" in comparison to Algorithm 1. Additionally, some of v's node-occurrences might have not received any pulse in this token

phase and they switch "early" compared to respective node in Algorithm 1 (i.e., before they receive the first DATA pulse). Nevertheless, they are all in the data phase when they need to send and receive DATA pulses. This essentially corresponds to Lines 8–17 in Algorithm 1. The same reasoning applies to every other node: once one of its occurrences receives the DATA pulse originated at $v*$ and switches to the data phase, then all its occurrences do so at the same time, but they all wait for the first DATA pulse to arrive (Line 34) and thus behave similarly to Algorithm 1, despite the "early" transition to the data phase.

Data Phase. *For the data phase, if node u has its $isTokenHolder_u$ set to* True*, then in Line 19 it dequeues a message from Q_u and denotes by 1^d its unary encoding. Next, in Lines 20-30, for each of the d pulses of* DATA *that need to be forwarded, each occurrence of u according to their order on C receives and sends the clockwise* DATA *pulse and then receives and sends the counterclockwise* END *pulse. This corresponds to Lines 19-23* in Algorithm 1, with two subtleties.

The main subtlety is that Line 19 is invoked only once by u, which corresponds to Line 19 in Algorithm 1 being invoked only by occurrence 0 of u in its rotation of C. This is essential, as otherwise if each occurrence of u initiates d separate DATA pulses then clearly there will be too many in the system and the message will not be correctly decoded. We emphasize that the queue Q_u is a single queue used by all occurrences of u, and hence once a message is dequeued from Q_u, other occurrences cannot dequeue it again later in further epochs if they become the occurrence of u that is associated with the $isTokenHolder_u$ variable being set to True.

The second subtlety is that each DATA pulse begins its traversal over C only after the previous one is received back at occurrence 0 of u. The latter does not harm the proof as it is only a stronger requirement compared to Algorithm 1.

Similarly, for every node u whose $isTokenHolder_u$ variable is set to False, each of its occurrences according to their order on C receives and sends the clockwise DATA pulse (in Lines 32–37) and the counterclockwise END pulse (in Lines 41–44), for d times. This corresponds to Lines 25 and 29 in Algorithm 1.

Finally, Lines 38-40 are also invoked by any node u only once, in order to avoid delivering to π duplicates of the received message, corresponding to Lines 26-28 in Algorithm 1. Note that this also means that every node u moves to the token phase of the next epoch in Algorithm 3(a) only after all of its occurrences finish the current data phase in Algorithm 3(b).

The above establishes that we can now repeat the proof of Lemma 7.3.1 for obtaining a proof of Lemma 7.4.1. ■

Lemma 7.4.1 allows proving the correctness of our simulation, as follows.

Theorem 7.7. *Let $G = (V, E)$ be some graph and let C be a Robbins cycle in it. Given any asynchronous protocol π, let $\hat{\pi}$ be the Algorithm 3 given the input π. Then, executing $\hat{\pi}$ on the fully-defective network G simulates an execution of π on the noiseless G.*

Proof The proof of Theorem 7.7 is exactly the same as the proof of Theorem 7.5, with the modifications that (i) it uses Lemma 7.4.1 instead of Lemma 7.3.1, (ii) instead of referring to a

node in the network, it refers to its occurences along C, and (iii) it adjusts the line numbers that reflect the delivery of a message to the protocol π (Line 40 in Algorithm 3(b) instead of Line 28 in Algorithm 1(b)). ■

Finally, the following lemma states the message overhead of our simulator. Its proof is identical to that of Lemma 7.3.2, except that we consider pulses sent by each of the *occurrences of nodes* on C, whose length $|C|$ can be greater than the number of nodes n.

Lemma 7.4.2. *Given a Robbins cycle C, the overhead of simulating a single message m in Algorithm 3 is $\mathsf{CC}_{\mathsf{overhead}}(m) = O(|C| \cdot 2^{|m|+O(\log n)})$.*

A direct application of the binary encoding described in Section 7.3.3 yields the following optimization.

Lemma 7.4.3. *Given a Robbins cycle C, the overhead of simulating a single message m in Algorithm 3, replacing the unary encoding with a binary encoding, is $\mathsf{CC}_{\mathsf{overhead}}(m) = O(|C| \cdot |m| + |C| \log n)$.*

We omit the details as they repeat the proofs in Section 7.3.3 for reducing the communication complexity in the simple cycle.

7.5 Constructing a Robbins Cycle in a Fully-Defective 2-Edge Connected Network

The simulator of Section 7.4 assumes the nodes are given a Robbins cycle on which they communicate. In this section, we show how the nodes can construct such a cycle on any 2-edge-connected fully-defective network G.

Whitney [Whi32] proved that any 2-edge-connected graph G can be decomposed into

$$G = C_0 \cup E_0 \cup E_1 \cup \cdots \cup E_k,$$

where C_0 is a simple cycle, and for any $i \geq 0$, E_i is an *ear*—a simple path or cycle whose endpoints belong to $C_0 \cup E_0 \cup \cdots \cup E_{i-1}$. Moreover, the process of decomposing G into ears can be performed by starting from a single node, and constructing C_0 and E_i in an increasing order $i = 0, 1, 2, \ldots$. See also [Lov85; KR00; Tsi04; Sch13] for further details and several distributed ear-decomposition algorithms in noiseless settings. Our Robbins cycle construction essentially performs a distributed and content-oblivious version of Whitney's ear-decomposition process (see, e.g., Lemma 2.1 in [Ram93] for the centralized algorithm), where nodes form the cycle C_0 and the ears $E_0, E_1, \ldots$ sequentially. A newly constructed ear is incorporated with the previous constructions to form a non-simple cycle that includes them all.

We start at a designated root node and perform a content-oblivious DFS by sending a token over edges in a sequential manner; see, e.g., [Pel00, Section 5.4]. This process continues

until the token returns to the root which signifies that a cycle is closed *at the root*. We require the constructed cycle to be *simple*. Indeed, if the token reaches some node $v \neq root$ twice, then v sends the token back to where it came from, which is equivalent to *backtracking* in the standard DFS algorithm. Backtracked edges do not participate in the constructed cycle, and they are left for future ears.

We denote the simple cycle constructed by the above procedure by C_0. The order the DFS-token progresses along C_0 defines the clockwise direction on the cycle. Nodes on C_0 employ Algorithm 3 to communicate over C_0 in a noise-resilient manner with the root being the first token holder.

Recall that a directed cycle can be represented by the nodes either locally, i.e., each node knows its clockwise and counterclockwise neighbor(s), or globally, i.e., knowing the sequence of IDs that defines the cycle. Our algorithm will use both representations, however, this is done only to simplify the analysis and reduce the length of the constructed Robbins cycle. In Remark 5 we sketch how to remove this assumption.

Before the nodes on C_0 continue with adding ears to C_0, they first broadcast their IDs and achieve a global representation of the cycle. The root sends its ID to its neighbor, who appends its own ID and transfers the message to its next neighbor and so on. When the message reaches the root again, it contains the sequence of IDs of the cycle $C_0 = (root, v_1, v_2, \ldots, root)$. The root broadcasts this information; it will be used towards continuing the Robbins cycle construction.

Next, the nodes on C_0 select a new root, denoted by $root_0$, to be one of the nodes on C_0 that still has *unexplored edges*, which are edges that do not participate in C_0. The construction proceeds by constructing a new ear, E_0, starting from $root_0$. Again, the nodes perform a sequential DFS by sending a DFS-token over unexplored edges, until the DFS-token reaches some node z_0 that belongs to C_0. As before, we require the path of the DFS-token to be simple, and backtrack whenever the token reaches twice the same node that does not lie on C_0.

The simple path that the DFS-token has undergone from $root_0$ to z_0, excluding edges that have backtracked in the DFS search, becomes the newly constructed ear E_0. A new ear can be a simple cycle if it is a closed ear with $z_0 = root_0$, or it can be a simple path if it is an open ear with $z_0 \neq root_0$.

Based on C_0 and the ear E_0, we define a new cycle C_1 that contains all the edges of C_0 and of E_0, possibly multiple times, so that C_1 is a closed (non-simple) cycle. Recall that in a Robbins cycle, each edge has a unique orientation and the cycle is not allowed to cross the same edge in both directions. Thus, we let C_1 be the cycle

$$root_0 \underset{C_0}{\longrightarrow} root_0 \underset{E_0}{\longrightarrow} z_0 \underset{C_0}{\Longrightarrow} root_0.$$

The notation $a \underset{P}{\longrightarrow} b$ here means that we take the complete path P. The notation $a \underset{P}{\Rightarrow} b$ means the *shortest path* from a to b implied by the clockwise orientation of edges in P. If multiple such paths exist, we take the first one by lexicographic order. Note that this path might not

be a sub-path of P,[1] however, it is uniquely defined and can be retrieved by any node that holds the sequence of IDs that defines P.

It follows that the paths $root_0 \underset{C_0}{\longrightarrow} root_0$ and $z_0 \underset{C_0}{\Longrightarrow} root_0$ are well defined and known by all nodes on C_0, since all these nodes know the sequence of IDs that lie on C_0, in their respective order. However, the nodes still need to know the IDs on $root_0 \underset{E_0}{\longrightarrow} z_0$ in order to obtain the sequence of IDs in the new cycle C_1. Towards this end we do the following.

The nodes on $P_0 \triangleq (z_0 \underset{C_0}{\Longrightarrow} root_0)$ along with the nodes on E_0 form a simple cycle $E_0 \| P_0$ (recall that $\|$ denotes concatenation). This cycle is locally defined: the nodes on E_0 define their neighbors when they first obtain the DFS-token. Each node on P_0 belongs to C_0 and, as argued above, can locally define its neighbors on P_0. Then, the root starts communicating over this cycle using Algorithm 3. As before, the first thing the nodes do is communicating their IDs. In fact, only the new nodes that are on E_0 but not in C_0 need to broadcast their IDs in their respective order, similarly to the way it was done after the completion of C_0. After this part, $root_0$ can simply construct the string of IDs of the nodes in C_1 and communicate it over $E_0 \| P_0$.

Then, the root communicates the sequence of IDs of C_1 to all the nodes in cycle C_0. That is, the root and the nodes on P_0 stop communicating on the cycle $P_0 \| E_0$ and switch back to communicating over the cycle C_0. Next, the root sends a message to instruct all the nodes in C_0 to switch to the new cycle C_1. Note that this message need not reach the nodes in E_0, as they are already "set" to the correct C_1. Since the other nodes are set to communicate over C_0, the nodes in E_0 are excluded from this communication and these nodes remain idle until the first pulse arrives, which happens once the rest of the nodes switch to communicate over C_1.

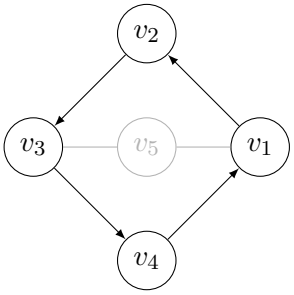
The process then repeats: for any $i > 0$, $root_i$ is selected to be a node on C_i that still has edges that do not belong to C_i. The nodes construct a new ear E_i whose endpoints, $root_i$ and z_i, belong to C_i. The nodes then locally define the non-simple cycle $C_{i+1} = root_i \underset{C_i}{\longrightarrow} root_i \underset{E_i}{\longrightarrow} z_i \underset{C_i}{\Longrightarrow} root_i$, and start communicating over it. Next, the nodes globally learn the sequence of IDs included in C_{i+1}, which is required for the next iteration, and so on. This process ends when the cycle C_{i+1} contains all the edges of G. See Figure 7.3 for a demonstration.

7.5.1 Formal Description

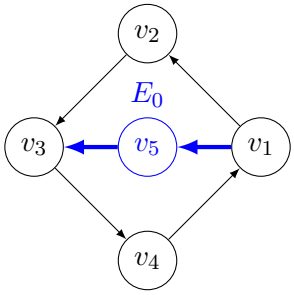
We now formally define our construction. Each node holds a variable named $cycle$ that contains a global representation of the current C_i. At the same time, the simple cycle $root_i \underset{E_i}{\longrightarrow} z_i \underset{C_i}{\Longrightarrow} root_i$ is represented locally, using the variables $next_v$ for the clockwise neighbor of v and $prev_v$ for its counterclockwise neighbor.

In our algorithms, the first ID in the variable $cycle$ is the current root. When the root node changes, each node v locally rotates the sequence of IDs in $cycle_v$ (say, clockwise), so that the new root becomes the first ID in the string.

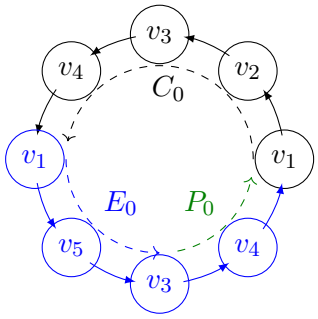
[1] For instance, let $P = (a \to b \to c \to b \to e)$, then $a \underset{P}{\Rightarrow} e$ is the path $(a \to b \to e)$ which is not a sub-path of P.



(a) An example of constructing a simple cycle, $C_0 = (v_1, v_2, v_3, v_4)$, starting from the root v_1. The clockwise direction corresponds to the pulse propagation in Algorithm 4(a) and is marked with arrows. The nodes set $next_{v_i} = v_{i+1}$ and $prev_{v_i} = v_{i-1}$ for $i = 1, 2, 3, 4$, except for $next_{v_4} = v_1$ and $prev_{v_1} = v_4$. Node v_5 does not receive any pulse and is not on C_0. Node v_5 keeps executing Algorithm 4(a), while the rest of the nodes continue to Algorithm 4(b).



(b) Adding the ear $E_0 = (v_1 \to v_5 \to v_3)$ to C_0. The nodes on C_0 execute Π_{NextRoot} to choose as a new root a node that has unexplored edges. Suppose v_1 is elected as the new root. The thick blue arrows describe the pulse propagation during the construction of E_0. The nodes v_1, v_5, v_3 update their *next* and *prev* to indicate this path. Then, nodes on $P_0 = (v_3 \to v_4 \to v_1)$, i.e., on $v_3 \underset{C_0}{\Longrightarrow} v_1$ update their *prev* and *next* to (locally) form the simple cycle $E_0 \| P_0$. This cycle is used to learn the IDs of nodes in E_0. The node v_1 constructs a global representation of C_1 and broadcasts it over C_0 and over $E_0 \| P_0$.



(c) The resulting Robbins cycle C_1, obtained after adding the ear E_0 described in part (b) to C_0. The clockwise direction of C_1 is marked with arrows.

Figure 7.3: Constructing a simple cycle by Algorithm 4(a) and extending an ear by Algorithm 4(b).

The pseudo-code for our content-oblivious protocol for constructing a Robbins cycle appears in Algorithms 4(a) and 4(b). These use as sub-procedures the protocols Π_{learnID} and Π_{NextRoot}, which are the content-oblivious versions of Algorithms 5 and 6, obtained by simulating them through Theorem 7.7. Note that all these algorithms share the same variables, i.e., $cycle_v$, $prev_v$, and $next_v$ of node v.

Our protocols use the ability to broadcast a message on a cycle defined either locally or globally. To be more accurate, the instruction "broadcast M" and "wait for message M" are to be understood as sending the message M with destination $*$ and receiving any message with destination $*$, respectively, using the method of Remark 3. The sender also receives the broadcast message after all other nodes receive it and acts upon the pseudo-code for processing it. This guarantees synchronization, i.e., that the sender does not continue before all other nodes receive the broadcast message, which is crucial, for example, when we switch the underlying cycle we communicate over. Indeed, in the noise-resilient protocol, the sender holds the token and does not release it before it gets the END pulse for that message, and by this time all other nodes receive that message as well. If now all nodes change their $cycle_v$, then the next pulse sent by the root goes through the new cycle.

7.5.2 Analysis

Our main theorem in this section shows that Algorithm 4 constructs a Robbins cycle that includes all the edges in G despite a fully-defective environment.

Theorem 7.8. *For any 2-edge-connected graph G, Algorithm 4 constructs a sequence of cycles $C_0, \ldots, C_k$, where C_0 is a simple cycle that includes the root, and C_k is a Robbins cycle that contains all the edges E of G.*

For the ease of the analysis, we define iterations of Algorithm 4. We say that iteration $i+1$ begins when the Π_{NextRoot} is being executed for the i-th time *by a node which is currently marked as root*, i.e., when such a node reaches either Line 33 or 64. Note that by the code, there can only be one root for each iteration. We start with some helping lemmas.

Lemma 7.5.1. *Suppose Algorithm 4(a) is executed by all nodes in a 2-edge-connected graph G, where a single node is marked as a root. Then, the root node eventually reaches Line 29, and at that time, there exists a single simple cycle C_0, locally represented by the nodes on it. Furthermore, $root \in C_0$.*

Proof It is immediate from the pseudo-code that Algorithm 4(a) performs a sequential depth first traversal starting from the root and using marked edges to avoid repeating already visited edges. We can think of the DFS as sending a DFS-token that progresses over non-visited edges until reaching a visited node v. The DFS-token advances by sending a single pulse.

Suppose the DFS-token reaches an already-visited node v, this node is either the root, in which case we are done, or it is not the root. In the latter case, the node v sends the DFS-token back to where it came from, causing the DFS to backtrack that edge and continue with the

Algorithm 4(a) Content-oblivious Ear-Decomposition: Closing an ear for the first time

Init: Set Π_{learnID} and Π_{NextRoot} to be the content-oblivious versions of Algorithms 5 and 6, respectively, obtained via Theorem 7.7.

node v**, upon initialization**:

$state_v \leftarrow$ init, $next_v \leftarrow \perp$, $prev_v \leftarrow \perp$, $cycle_v \leftarrow \epsilon$. All edges unmarked.
if v is the root **then**
    choose an arbitrary edge (v, u)
    send a pulse to u and mark the edge (v, u) as used.
    $next_v \leftarrow u$, $state_v \leftarrow$ DFSroot
end if

node v**, upon receiving a pulse from** w:

if $state_v =$ init **then**
    $prev_v \leftarrow w$, mark (w, v) as used
    choose an arbitrary neighbor $u \neq w$ where (v, u) is unmarked
    send a pulse to u and mark (v, u) as used
    $next_v \leftarrow u$, $state_v \leftarrow$ DFS
else if $state_v =$ DFS **then**
    if $w = next_v$ **then** ▷ This is a cancellation pulse
        choose an arbitrary neighbor u' where (v, u') is unmarked:
            send a pulse to u', set $next_v \leftarrow u'$ and mark (v, u') as used
        if no such u' exists **then**
            send a pulse to $prev_v$ ▷ Send a cancellation pulse to parent
            $state_v \leftarrow$ init, $prev_v \leftarrow \perp$, $next_v \leftarrow \perp$, unmark all edges
        end if
    else if $w \neq prev_v$ **then** ▷ A cycle is closed at v, but v is not the root
        send a pulse to w and mark (v, w) as used.
    else $(w = prev_v)$ ▷ This is a second pulse—node is on a cycle
        send a pulse to $next_v$
        $cycle_v \leftarrow \Pi_{\mathsf{learnID}}$, executed over the cycle locally defined by $prev_v$, $next_v$; initialize as non token holder.
        execute Π_{NextRoot} over $cycle_v$; initialize as non token-holder.
    end if
else if $state_v =$ DFSroot **then**
    $prev_v \leftarrow w$
    send a pulse to $next_v$ ▷ A cycle is closed, start communicating on it
    wait until a pulse is received from $prev_v$
    $cycle_v \leftarrow \Pi_{\mathsf{learnID}}$, executed over the simple cycle locally defined by $prev_v, next_v$; initialize as token holder.
    execute Π_{NextRoot} over $cycle_v$; initialize as token holder.
end if

Algorithm 4(b) Content-oblivious Ear-Decomposition: Ear extension

node v **marked as** $root$**, upon initialization**:

35: choose an edge $(v, u) \notin cycle_v$ and send a pulse to u

36: $next_v \leftarrow u$

node v**, upon receiving a pulse on** $(v, u) \notin cycle_v$:

37: $prev_v \leftarrow u$

38: broadcast "$\langle \mathsf{EarClosedAt} \rangle, v$" over $cycle_v$

▷ In parallel to the above, pulses from $cycle_v$ are interpreted as messages of a noise-resilient protocol

node v**, upon receiving "**$\langle \mathsf{EarClosedAt} \rangle, w$**" on** $cycle_v$:

39: $P_i \leftarrow$ the simple path $w \xrightarrow[cycle_v]{} root$ ▷ $P_i = \emptyset$ if $w = root$

40: **if** $v \in P_i$ **then**

41: set $prev_v$, $next_v$ according to P_i ▷ The root sets $prev$ and w sets $next$ (unless $root = w$); inner nodes set both

42: **end if**

43: **if** v is the root **then**

44: send a pulse to $next_v$

45: **if** $root = w$ **then**

46: wait to receive a pulse from $prev_v$ ▷ A closed ear, the pulse will reach back the root

47: broadcast $\langle \mathsf{ready} \rangle$ on $cycle_v$

48: **end if**

49: **else if** $v = w$ **then** ▷ $w \neq root$

50: wait to receive a pulse from $prev_v$

51: broadcast $\langle \mathsf{ready} \rangle$ on $cycle_v$

52: **end if**

53: wait to receive $\langle \mathsf{ready} \rangle$ on $cycle_v$

54: **if** $prev_v, next_v \neq \perp$ **then** ▷ v is on P_i

55: execute Π_{learnID} over the simple cycle locally defined by $prev_v, next_v$; root is token holder.

56: $prev_v \leftarrow \perp$, $next_v \leftarrow \perp$

57: **end if**

58: **if** v is the root **then**

59: broadcast "$\langle \mathsf{NewCycle} \rangle, C_{i+1}$" over $cycle_v$, where C_{i+1} is the output of Π_{learnID}.

60: **else**

61: wait to receive the message "$\langle \mathsf{NewCycle} \rangle, C_{i+1}$" over $cycle_v$.

62: **end if**

63: $cycle_v \leftarrow C_{i+1}$ ▷ All nodes in C_i switch to C_{i+1}; nodes on E_i were set at line 25

64: execute Π_{NextRoot} over $cycle_v$; The root initializes as the token holder

Algorithm 5 π_{learnID}, learning the IDs on a newly constructed ear (noiseless setting)

node v**, upon initialization:**

1: **if** v is the root **then**
2: send $id(v)$ to $next_v$
3: **end if**

node v**, upon receiving** $m = (id_1, id_2, \ldots)$:

4: **if** $id_1 \neq id(v)$ **then** ▷ $\{next_v\}_{v \in V}$ is guaranteed to induce a simple cycle
5: $m' \leftarrow m \| id(v)$
6: send m' to $next_v$
7: **else** ▷ Back to root, m contains all the nodes on $\{next_v\}_{v \in V}$
8: $new\_cycle \leftarrow cycle_v \| m$
9: broadcast "$\langle \mathsf{done} \rangle, new\_cycle$"
10: **end if**

node v**, upon receiving** "$\langle \mathsf{done} \rangle, C$":

11: return C

DFS from the parent of v in the induced DFS tree. Since the graph is 2-edge-connected, there exists a simple cycle that begins and ends at the root. A DFS search, once completed, explores all the edges in G. Therefore, the DFS must eventually reach the root again and close a *simple* cycle, defined by the progress of the DFS-token while ignoring any backtracked edges. Indeed, each node sets its $prev_v$ variable to the first node from which the DFS-token is received and sets its $next_v$ variable to be the node to which the DFS-token progresses. Backtracking an edge resets $prev_v$, $next_v$, accordingly in Lines 16 or 19.

Denote the above constructed cycle as C_0. We note that nodes that are not on C_0 are either never reached by the DFS or the DFS reaches them and backtracks since it does not reach the root from that path. In either case, their status at the time when the root reaches Line 29, and also at the end of Algorithm 4(a), is $init$ with no marked edges, and with $prev = next = \perp$. Therefore, C_0 is the only cycle defined at this point. ■

Next, we observe that the nodes on C_0 switch to a global representation of their cycle.

Lemma 7.5.2. *Once the root completes Line 32, all the nodes on C_0 hold a global representation string of C_0.*

Proof Lemma 7.5.1 establishes that once the root reaches Line 29, then C_0 is locally well-defined, i.e., every node that belongs to C_0 knows the previous and subsequent nodes in the cycle. The root then sends a second pulse which progresses over C_0 and causes all the nodes on C_0 to execute Π_{learnID}, where the root is the token holder (Line 32) and other nodes are non token holders (Line 25). Note that the root awaits until the second pulse reaches it back (Line 31). By that time, all the other nodes on C_0 start executing Π_{learnID}, but they are not

Algorithm 6 π_{NextRoot}, choosing a new root (noiseless setting)

node v**, upon initialization:**
1: **if** v is the root **then**
2: broadcast "⟨check edges⟩"
3: wait to receive $|\{id(v') \mid v' \in cycle_v\}|$ many replies
4: **if** received "⟨has unexplored edges⟩, $id(u)$" **then** ▷ Choose arbitrarily, if non unique
5: broadcast "⟨new root⟩, $id(u)$"
6: **else** ▷ All edges are explored
7: broadcast "⟨completed⟩"
8: **end if**
9: **end if**

node v**, upon receiving** ⟨check edges⟩:
10: **if** v has unexplored edges **then**
11: broadcast "⟨*has unexplored edges*⟩, $id(v)$"
12: **else**
13: broadcast "⟨no unexplored edges⟩, $id(v)$"
14: **end if**

node v**, upon receiving** "⟨new root⟩, $id(u)$": ▷ Broadcast message is received also by its originator
15: rotate $cycle_v$ clockwise until it starts with an occurrence of u. The node u is now marked root
16: execute Algorithm 4(b)

node v**, upon receiving** ⟨completed⟩:
17: terminate ▷ A Robbins cycle is constructed

token holders, so they remain idle. Only once the root starts executing Π_{learnID}, pulses are sent over C_0 and the content-oblivious computation of Algorithm 5 initiates.

The execution of Algorithm 5 produces the sequence of IDs in C_0 according to the clockwise direction of the cycle: the root begins by sending its ID to its $next$ (clockwise) neighbor, which concatenates its ID, and so on. Once the message reaches the root again, it contains all the IDs of the nodes in C_0 *according to the clockwise direction of the cycle.* This string is then broadcast to all C_0, so all the nodes now possess the global representation of C_0 as required.■

Note that after the construction of C_0 completes, the nodes that belong to C_0 continue to execute Algorithm 4(b), while the rest of the nodes are still executing Algorithm 4(a). We now argue that the algorithm keeps adding edges to the currently-constructed cycle.

For a cycle C, let us denote by $Edge(C)$ the set of edges in C. We prove that each iteration of Algorithm 4 constructs a larger cycle. That is, assuming the nodes on C execute Algorithm 4(b) while the rest of the nodes execute Algorithm 4(a), then at the end of that

iteration, there is a globally defined cycle C' such that all the nodes on C' know this cycle (the other nodes keep executing Algorithm 4(a)), and C' is strictly larger than C, that is, $Edge(C) \subsetneq Edge(C')$.

Lemma 7.5.3. *Let G be a 2-edge-connected graph and let C_i be a cycle, such that $E \backslash Edge(C_i) \neq \emptyset$. Let* the root *be a single marked node on C_i that is adjacent to an edge in $E \backslash Edge(C_i)$. Suppose nodes on C_i all start executing Algorithm 4(b) while other nodes in G run Algorithm 4(a) and their state is $init$. At the end of this iteration, there exists a cycle C_{i+1} with $Edge(C_i) \subsetneq Edge(C_{i+1})$, all the nodes on C_{i+1} know its global representation, and all the other nodes continue executing Algorithm 4(a) and their state is $init$. Further, if all the occurrences of any edge in $Edge(C_i)$ have the same orientation, the same holds for C_{i+1}.*

Proof Note that the nodes basically perform a DFS search over the unused edges, i.e., over all the edges except edges that belong to C_i. The $root$ initiates the DFS search (Line 35). Since the root has at least one edge which does not belong to C_i, denote the edge to which the root sends a pulse in Line 35 by $(root, v)$.

We argue that the DFS, after passing the DFS-token over $(root, v)$, must reach a node that belongs to C_i before it backtracks the edge $(root, v)$. Suppose not, then there is no path between v and any node in C_i that does not go through $(root, v)$. Hence, $(root, v)$ is a bridge, yet this is a contradiction since G is 2-edge-connected.

Once the DFS reaches some node z on C_i in Line 37, the path E_i is well defined: it is the new ear—the path the token has taken from $root$ to z, disregarding any backtracked edge. Note that E_i is not empty and $Edge(E_i) \subseteq E \setminus Edge(C_i)$, i.e., E_i contains at least one new edge that does not belong to C_i. Additionally, the path P_i constructed in Line 39 is well defined: it is the shortest path between z and $root$ that uses only the directed edges in $Edge(C_i)$. We know at least one such path exists since z and $root$ are both nodes on the cycle C_i, and take the lexicographic-first such path if multiple shortest-paths exist. Since all nodes on C_i know $Edge(C_i)$ then P_i is agreed upon all of them. Hence $C_{i+1} = C_i \| E_i \| P_i$ is a well defined cycle from $root$ to $root$ for which $Edge(C_i) \subsetneq Edge(C_{i+1})$. It is easy to verify that all the occurrences of any edge in $Edge(C_{i+1})$ have the same orientation: edges in E_i appear only once in C_i, and all the other edges obey their orientation in C_i, which is unique by assumption.

We now show that at the end of the iteration, all the nodes on C_{i+1} hold a global representation of C_{i+1} while the rest of the nodes remain in state $init$, executing Algorithm 4(a). Note that as the DFS progresses through E_i, all the nodes on E_i define their $next$ and $prev$ variables according to the progress of the DFS-token, so that the path E_i is locally defined. After the DFS-token reaches z in Line 37, this node communicates over C_i to let all the nodes of C_i know that an ear is closed and its endpoints are $root$ and z. With this information, each node on C_i can tell whether it belongs to P_i, and if it is on P_i, it can tell its successor and predecessor nodes on P_i. Thus, each such node locally sets its $next$ and $prev$ variables according to the path P_i in Line 41. Note that the concatenation of the two paths, $E_i \| P_i$, yields a

simple cycle, locally defined by all the nodes on it. Also note that if E_i is a closed ear, when $z = root$, then $P_i = \emptyset$, yet $E_i \| P_i$ is still a simple cycle.

Next, the root sends a second pulse in Line 44 which propagates along E_i and triggers the nodes on E_i, except for $root$ and z, to start executing Π_{learnID} on the cycle locally defined by their $next$ and $prev$ variables (Line 25). However, none of the (inner) nodes on E_i is the token holders in the execution of Π_{learnID}, so they remain idle, in the sense that they do not request the token.

Once this second pulse reaches z in Line 51, it informs the nodes in C_i about this event by broadcasting $\langle \mathsf{ready} \rangle$ on C_i. Note that at this point, the nodes on C_i are all idle. Specifically, no node wishes to obtain the token, so no pulses are being sent over C_i. It is safe to switch to communicating over the locally defined simple cycle $E_i \| P_i$. The nodes on that cycle now execute Π_{learnID}, after which all of them learn the global string representing $C_{i+1} = C_i \| E_i \| P_i$. At this point, the nodes in E_i except $root$ and z switch to communicate over C_{i+1}. However, they are not the token holders so they keep being idle until the rest of the nodes switch to C_{i+1}, without interfering with them.

After Π_{learnID} terminates, all the nodes on $E_i \| P_i$ that were executing it know it has terminated. The root is the last to obtain the final message "$\langle \mathsf{done} \rangle, C_{i+1}$", so at the time when the root finishes Π_{learnID}, all other nodes on C_i are set to communicate over C_i: the nodes on P_i are done with Π_{learnID}, and set $next = prev = \perp$ in Line 56, and now await the $\langle \mathsf{NewCycle} \rangle$ message *on* C_i. The rest of the nodes on C_i do not perform the **if** statement of Line 54 and thus are already awaiting the $\langle \mathsf{NewCycle} \rangle$ message.

Finally, the root broadcasts "$\langle \mathsf{NewCycle} \rangle, C_{i+1}$" over C_i which causes all the nodes in C_i to change their $cycle$ variable to C_{i+1}. The root is the last to finish the procedure of the broadcast invocation, and by that time, all nodes of C_{i+1} are set to the cycle C_{i+1} and idle. The root is the token holder and is expected to send the next message on C_{i+1}. ■

The proof of Theorem 7.8 can now easily be obtained as a corollary of the above lemma. Multiple invocations of Algorithm 4(b) eventually yield a Robbins cycle C_k with $Edge(C_k) = E$.

Proof of Theorem 7.8 By Lemma 7.5.1, we know that after the first iteration of Algorithm 4(a) we obtain a simple cycle C_0. If C_0 consists of all the edges of G, we are done—the nodes run Π_{NextRoot} to find out that all edges are exhausted, and the algorithm terminates in Line 17 of Algorithm 6. Otherwise, we keep executing Algorithm 4(b) with a new root that has an adjacent unused edge. This is done by Algorithm 6: each node broadcasts whether or not it has unused edges adjacent to it, along with its ID. The current root arbitrarily picks one node with unused edges (Line 4) and broadcasts this choice to all the nodes of C_i. Since all the nodes possess a global representation of C_i, they can rotate it so that the new root becomes first in the global representation, which is consistent among all nodes and allows, for example, to determine P_i in a consistent manner. Then, Algorithm 4(b) is invoked again with this chosen node as the new root (Line 16). At this point, the statement of Lemma 7.5.3 holds: there is a cycle C_i globally represented by all the nodes in it, there is a single root on C_i

and it has adjacent unused edges, and all the nodes in $G \setminus C_i$ are in state $init$ in the execution of Algorithm 4(a).

By Lemma 7.5.3, every iteration of the algorithm starting on C_i produces a cycle C_{i+1} with at least one additional edge in E that does not appear in C_i. It is easy to verify that, as long as some edge is still unused, at the end of constructing C_{i+1}, i.e., after executing Line 64 but before the nodes re-iterate Algorithm 4(b) (Line 16 of Algorithm 6), the requirements for Lemma 7.5.3 hold with respect to the newly constructed cycle. Thus, after at most $|E| - |Edge(C_0)|$ iterations of Algorithm 4(b), the obtained cycle consists of all the edges E in G. Since each edge has a single orientation induced by the cycle (this clearly holds for the simple cycle C_0, and inductively throughout the construction), and since all the nodes in G appear in the obtained cycle, it is a Robbins cycle. ■

Remark. In order to communicate over any intermediate (non-simple) cycle C_i via Algorithm 3, a single node-occurrence must be defined as the token holder. Furthermore, all other nodes must know the segment in C_i that contains that designated node-occurrence. Recall that in Algorithm 3, each node maintains the invariant that the token resides in its segment 0 (see Section 7.4). Our construction indeed provides the nodes with this information, which can be retrieved from the global representation of C_i. The first node-occurrence in C_i is defined to be the token holder, and each other node can re-number its occurrences along C_i in the natural manner, so it is consistent with having the token at its segment 0. The above also holds also for the Robbins cycle C_k constructed in Theorem 7.8.

Remark. **Avoiding Global Knowledge:** In the above construction, the nodes obtain a global representation of the cycles C_i they construct. We remark that this knowledge helps in simplifying the construction and reducing the length of the constructed cycle. However, it is not necessary, and a similar construction can be designed in which each node only holds local information about C_i, i.e., only its clockwise and counterclockwise neighbors for each of its occurrences on C_i. We provide here the main differences in such a construction.

(1) The global representation of C_i is used to determine the path P_i between the end points $(root, z)$ of the newly constructed ear E_i. For the above construction to work, we need every node to know whether or not it belongs to P_i; if it is part of P_i, then it should appear one more time in C_{i+1}. Now, suppose that every node v on C_i knows only a local representation of C_i, namely, its $next$ and $prev$ neighbors for each occurrence of v on C_i. The path P_i can be determined in the following way. Once the endpoint z of the ear E_i broadcasts the message "$\langle \mathsf{EarClosedAt} \rangle, z$" over C_i, all the nodes in C_i switch to a new state of "detecting P_i". In this state, if a node-occurrence receives a clockwise pulse, it means that this occurrence belongs to P_i. A counterclockwise pulse signifies that the node-occurrence should quit this new state and continue executing Algorithm 4(b). In both cases, each pulse is propagated by the node-occurrence along the same direction it is received.

The nodes use the above mechanism as follows. Once the broadcast of "$\langle \mathsf{EarClosedAt} \rangle, z$" completes at z, it sends a single clockwise pulse. This pulse propagates along C_i until it reaches a node-occurrence of the root; denote by P_i the path that this pulse has taken. The

root *does not* propagate the pulse, but instead sends a single counterclockwise pulse, which travels along the entire C_i until reaching that same root node-occurrence again. At this point, all the node-occurrences that belong to P_i have received a clockwise pulse, and all the node-occurrences on C_i have received a counterclockwise pulse, so all nodes can continue with the construction as above. Note that this method also allows the nodes to track the segment in which the root lies, so that at the end of the construction they can infer the token segment at any step.

(2) The other place our construction uses the global representation is in π_{NextRoot}, where the root awaits to receive a message from every node on C_i to know whether the construction is done. However, without a global representation, the root does not know how many nodes are in C_i and thus it cannot know how many messages to expect. The remedy for this issue utilizes the token delivery method of Algorithm 3. Namely, we replace Algorithm 6 with the following method. The root begins by broadcasting $\langle$check edges$\rangle$. Every node that still has an unexplored edge requests the token, and if it receives the token, it sends its ID. The first node to do so becomes the new root. If no such node exists, the token propagates until it reaches the (old) root again. In this case, the root acquires the token and broadcasts $\langle$completed$\rangle$ to indicate that the Robbins construction is done.

Remark. **Coping with KT_0:** Algorithm 5 and its noise-resilient form Π_{learnID} are KT_1 algorithms, in which each node knows the IDs of its neighbors. We remark that we can establish the learn-ID functionality, and thus the construction of the Robbins cycle, even in KT_0 networks, in which the IDs of the neighbors of a node are not known to it upon initialization. Note that Algorithm 5 as stated cannot work in a KT_0 network since a node does not know which node comes immediately next to it in the cycle. In other words, after the root sends its ID as the first message, this message reaches *all* other nodes and none of them knows they are the next one on C_0.

We can solve this issue by relying on the order in which the token holder shifts in the underlying simulator. A KT_0 protocol for learning the IDs starts by instructing all the nodes to broadcast their ID. Thus, all nodes request to be token holders. Once the root sends its own ID and releases the token, its immediate *counterclockwise* neighbor becomes the new token holder. Thus, the IDs are broadcast exactly in their counterclockwise order on C_0. Once the root becomes a token holder again, this process is done.

We also note that the simulator of Section 7.4 only requires local knowledge of a Robbins cycle and thus can run on KT_0 networks with the above pre-processing step. Thus, Theorem 7.2 holds for KT_0 networks as well.

7.5.3 The Length of the Obtained Robbins Cycle

We complete this section with a crude analysis of the size of Robbins cycle our construction obtains and the communication complexity of the construction.

Lemma 7.5.4. *Let G be a 2-edge-connected graph, and let C be the Robbins cycle constructed by Theorem 7.8. Then $|C| = O(n^3)$. Further, Algorithm 4 communicates $O(n^8 \log n)$ pulses altogether.*

Proof Given some C_i, it holds that $|C_{i+1}| = |C_i| + |E_i| + |P_i|$. Since P_i is a shortest (simple) path between two nodes, we have $|P_i| < n$, for all iterations i. A bound on the worst-case length of the Robbins cycle is obtained by considering $O(n^2)$ iterations of Algorithm 4, in each of which, adding only a single edge to the current C_i. In this case, the cycle's length extends by $O(n)$ in each of the $O(n^2)$ iterations, yielding a total length of $O(n^3)$.

Let us now bound the communication complexity. Consider the iteration where the nodes begin with C_i and construct C_{i+1}. The π_{learnID} algorithm communicates at most $\alpha_i = |E_i| + |P_i|$ messages, each of length at most $O(\alpha_i \log n)$, except for the $\langle \mathsf{done} \rangle$ message whose length is $O(|C_{i+1}| \log n)$. The π_{NextRoot} algorithm communicates $|C_{i+1}|$ messages of length $O(\log n)$. The rest of Algorithm 4(b) makes $O(1)$ broadcasts of messages of length $O(\log n)$, and a single $\langle \mathsf{NewCycle} \rangle$ message whose length is α_i. Recall that by Lemma 7.4.3, broadcasting a message of length m over the cycle C_i takes $O(|C_i|(m + \log n))$ pulses.

Next, we argue that the DFS search within a single iteration of Algorithm 4 sends $O(n^2)$ pulses. To see that, recall that each edge is marked as used once the DFS-token passes through it. Additionally, the token might backtrack that edge, but no more pulses should be sent on that edge, leading to a total of at most $2|E| = O(n^2)$ pulses overall. The above does not hold for nodes that have backtracked all their edges and reset their state to $init$, because they also unmark all their edges and might re-send pulses over edges that were already explored in this iteration. We argue, however, that such nodes will never get the DFS-token again during that iteration. Indeed, assume towards contradiction that u is a node that has reset its state during the current iteration and is *the first* node that receives the DFS-token after resetting its state, say, over the edge (u, v). Since u has explored and backtracked all its edges, the DFS-token must have already passed through the edge (u, v) previously in this iteration. Therefore, it is marked used by v, and it is impossible that v sends a DFS-token over this edge, unless v resets its state and unmarks all its edges. However, if v reset its state and then sends a DFS-token over (u, v), then v must have received the DFS-token after resetting and before u did, contradicting our choice of u.

We then conclude that the complexity of constructing the Robbins cycle in Algorithm 4 is bounded by

$$\sum_i \Big[\alpha_i \cdot O(\alpha_i \cdot \alpha_i \log n) + O(\alpha_i \cdot |C_i| \log n) + |C_{i+1}| \cdot O(|C_{i+1}| \log n) + O(|C_i| \log n) + O(n^2)\Big]$$

pulses. Bounding $\alpha_i = O(n)$ and $|C_i|, |C_{i+1}| = O(n^3)$, and the number of iterations $i \leq |E| = O(n^2)$, we conclude that the complexity of constructing the Robbins cycle is $O(n^8 \log n)$ pulses. ■

Note that the complexity can be reduced if we assume KT_1 networks and global representation of the constructed cycle. Instead of terminating when all the adjacent edges of all the

nodes were explored, we terminate when all nodes see that all their neighbors appear on the current C_i. Each node can determine this information assuming KT_1 knowledge and a global representation of the cycle. This guarantees that at least one node is added at each iteration of Algorithm 4, which reduces the number of iterations to $i \leq n$. This method leads to a Robbins cycle of total length $O(n^2)$ and a communication complexity of $O(n^6 \log n)$.

7.6 Impossibility of Resilient Communication in Fully-Defective Networks which are not 2-Edge Connected

In this section we complement our simulator for 2-edge-connected graphs, with a proof showing that 2-edge connectivity is required for communication in fully-defective networks. The intuitive argument is that if the communication network is not 2-edge connected, then a bridge exists, and corrupting messages over that edge will lead to disconnecting the network, preventing the correct computation of any non-trivial function. Towards that goal we show the impossibility of asynchronous computation with *two parties* in the presence of fully-defective channel noise. The two-party impossibility implies a general impossibility result for any network that contains a bridge since the two connected components over the two sides of the bridge can be reduced to the two parties case.

Formalizing the above intuition is slightly more subtle. For the impossibility to hold, we must require the protocol to give output (or explicitly terminate). To see why, consider the case of two parties (say, Alice and Bob) that hold the private inputs x and y, respectively, and need to compute some fixed known function $f(x, y)$. Suppose that, instead of requiring the protocol to give a non-revocable output, we only require that there exists a time t after which both parties hold $f(x, y)$ and never change it again. Then, the following protocol succeeds in computing f in the fully-defective two-party network (stated for Alice; Bob's protocol is symmetric): (a) Send x messages to Bob; (b) $\mathsf{count} \leftarrow 0$; (c) Upon the reception of a message, $\mathsf{count} \leftarrow \mathsf{count} + 1$; update the output variable to $f(x, \mathsf{count})$.

Nevertheless, if we require the parties to terminate or to give an output, no protocol for non-trivial functions f exists.

Theorem 7.9. *Consider a fully-defective network of two parties connected via a single noisy channel, and let $f(x, y)$ be any non-constant function. Any two-party deterministic protocol that computes f and gives an output, is incorrect.*

Proof Let f be some non-constant function and assume, without loss of generality, that its input and output domains are the natural numbers. We can restrict the discussion to protocols in which each message sent by any of the parties contains a single '1' bit. This is without loss of generality, since we can equivalently consider the case where the adversary corrupts the content of any message to be '1'. Since the setting is asynchronous, a party can send zero or more messages as a function of its input and the number of messages it has received so far. A party is assumed to be idle between the time it sends a batch of messages until the time

a new message arrives (which may trigger the transmission of new messages). In particular, once a new message arrives, the party immediately decides upon the number $k \geq 0$ of new messages to send, transmits them, and then goes back to being idle (or terminates).

Consider some inputs (x, y) and (x', y) for which $f(x, y) \neq f(x', y)$, if no such inputs exist then a symmetric proof holds for a pair of inputs (x, y) and (x, y'). Fix Bob's input to y. Note that once y is fixed, Bob's actions depend only on the number of messages he has received so far. That is, we can completely describe Bob's protocol by the sequence $\mathcal{B}_y = (0, \mathsf{action}_0)(1, \mathsf{action}_1)(2, \mathsf{action}_2)\cdots$, where for any $t \geq 0$, the item (t, action_t) is to be interpreted as the action Bob performs after seeing t messages from Alice. The value $\mathsf{action}_t \in \{\mathsf{send}_k, \mathsf{SendAndOutput}_{k,r}\}_{k,r\geq 0}$ describes the action Bob takes at that step of the protocol: send_k means that Bob transmits k messages to Alice, and $\mathsf{SendAndOutput}_{k,r}$ means that Bob sends k messages to Alice and sets its output register (irrevocably) to r, i.e., Bob commits to the output r. Note that this is a complete characterization of Bob's protocol. We may assume that Bob continues to send and receive messages after setting its output, however, if in a later step Bob performs the action $\mathsf{SendAndOutput}_{k,r}$, then Bob will only send k messages but the output register will not change.

Also note that Bob progresses sequentially. That is, Bob first performs action_0, then action_1, etc. Once Bob receives no further messages from Alice, he stops making any further progress. Thus, in order to give an output, Bob must reach some $t \geq 0$ where $\mathsf{action}_t = \mathsf{SendAndOutput}_{k,r}$. Consider $\mathcal{B}_y$ and set $\hat{t} = \arg\min_t(\mathsf{action}_t \in \{\mathsf{SendAndOutput}_{k,r}\}_{k,r\geq 0})$; we know that $\hat{t} < \infty$ and $\mathsf{action}_{\hat{t}} = \mathsf{SendAndOutput}_{\hat{k},\hat{r}}$, with some $\hat{k}, \hat{r} \geq 0$, or otherwise Bob never gives an output on input y. Finally, we note that Bob acts as described regardless of Alice's input: Bob advances sequentially until seeing $\hat{t}$ messages from Alice, after which it commits on the output $\hat{r}$.

Now consider an execution of the protocol on the input (x, y). As described above, Bob commits on output when performing $\mathsf{action}_{\hat{t}} = \mathsf{SendAndOutput}_{\hat{k},\hat{r}}$. If Bob does not give the correct output, we are done. Otherwise, $\hat{r} = f(x, y)$. Next, consider the execution of the protocol on the input (x', y). If Bob receives less than $\hat{t}$ messages overall (and the protocol then reaches quiescence), Bob does not give an output. Otherwise, upon receiving the $\hat{t}$-th message, Bob outputs $\hat{r} = f(x, y)$. As both these options are incorrect for the input (x', y), we have reached a contradiction. ■

7.7 Conclusion and Open Questions

We showed that *content-oblivious computation*, where message content is empty or invalid, is possible in any 2-edge connected network. Further, this condition is necessary: No non-trivial computation can be performed in networks that are not 2-edge connected and have a bridge. Content-oblivious computation may have applications in systems suffering from very harsh noise or in systems where communication is extremely limited, so nodes can only signal each other by sending pulses.

We conclude with some open questions and research directions.

- **Overhead.** Our main motivation was to show the feasibility of content-oblivious simulation, and we did not care too much about the resulting overhead caused by our compiler. Finding the minimum overhead for content-obvious simulations and the exact relationship between topology properties and overhead is an interesting open question. For example, any graph that can be decomposed into (short) disjoint cycles will have overhead proportional to its longest cycle length. Can other properties be exploited to achieve a faster compiler, for example, when the network is k-edge-connected, with $k > 2$? Is it possible to simulate computations using an infrastructure other than cycles, such as (a family of) spanning trees? The overhead of the pre-processing phase can potentially be further reduced. As mentioned in Section 7.5.3, our Robbins Cycle construction exhausts all edges in the network, even when an edge does not add new nodes to the cycle. Constructing cycles over directed graphs or when edges have weights (which we try to minimize) might lead to compelling applications.

- **A root.** The preprocessing phase of our construction assumes that a particular node has been preselected as the root. We conjecture that this assumption is necessary for any non-trivial content-oblivious computation.

- **Probabilistic protocols.** Our construction assumes deterministic algorithms. If the noiseless protocol is probabilistic, then our compiler will still simulate one specific execution of the noiseless protocol. However, this will affect the success probability and might have further undesired consequences. Extending the simulation and analysis to probabilistic algorithms remains an interesting open direction.

- **Content-oblivious computation of specific tasks.** Our compiler can be used to compute any task (that has a noiseless protocol) in a content-oblivious manner, but with high overhead. Can some specific tasks be solved more efficiently *directly* (i.e., without applying our general compiler)? In this work, we gave direct constructions for Robbins Cycle, DFS, and ear-decomposition. In [CGH19], a direct BFS construction was given. What other tasks can be computed (fast!) in this setting?

Chapter 8

Conclusion

This thesis explores concurrent and distributed systems guaranteeing efficiency, correctness, and robustness. We pushed the boundaries of what is possible and introduced novel algorithms that improve upon previous literature: the first methodology for an efficient and correct concurrent size operation, optimal efficient concurrent durable FIFO queues for NVRAM, and content-oblivious protocols for computations in fully-defective networks.

In a broader sense, we enhanced our understanding of the limitations of computation in concurrent and distributed systems. For instance, we identified which network graphs could tolerate unbounded alteration noise, and proved that the desirable performance and robustness properties we propose for parallel distributed transactional systems cannot be achieved all together.

The research presented in this thesis opens various avenues for future exploration. Firstly, our possibility and impossibility results could guide the development of future improved implementations within the feasible design space. As an example, content-oblivious computation techniques for 2-edge connected networks could be enhanced, e.g., by targeting a specific more limited topology. Another example is designing PDTSs that satisfy a subset of the performance properties we propose. Secondly, studies similar to ours, exploring variations of the models and properties we addressed, could lead to new impossibility results and new algorithms for relevant applications. Examining the impact of stronger progress conditions for parallel distributed transactional systems on performance properties is one potential direction. Another is developing correct and efficient algorithms for data structures' aggregate properties other than size. A summary of our contributions, along with further directions for future research, follows.

Correctness

We addressed the challenge of obtaining the size of concurrent data structures while maintaining both correctness and efficiency in Chapter 3. Even though the size is an elementary property, existing library implementations are incorrect, returning an approximation of the size. Our proposed methodology for adding a linearizable size operation to concurrent sets and dictionaries has attractive theoretical properties in terms of progress guarantees and asymp-

totic time complexity. Evaluation of correct size computations demonstrated that while our methodology incurs some overhead on the original operations of the data structures, it significantly improves the performance of the size operation—by orders of magnitude compared to existing solutions—and yields a scalable size operation, with performance insensitive to the size of the data structure. Reducing the overhead on the original operations remains the missing piece to making the methodology fully acceptable for data structures in concurrent libraries in Java, Python, C++, etc. Other than the size property, other kinds of range queries would benefit from a correct and efficient solution, that would return succinct information about the input range of data items without taking a snapshot of the target range and traversing its elements. This direction was explored in recent works [KAA24; SP24; FR24].

Linearizability is a widely accepted correctness condition for concurrent objects. In Chapter 4 we drew attention to a somewhat-neglected aspect of linearizability concerning the handling of pending invocations, particularly relevant for systems with non-volatile main memory. We identified a typo in the original definition of linearizability which impacts this aspect, and provided an amendment to correct the definition. We believe that rigorously addressing this issue is important and timely.

Performance

Inspired by recent advancements in network capabilities, we studied in Chapter 5 distributed transactional systems that leverage server parallelism. We formalized performance properties of distributed transactional systems—distributed disjoint-access parallelism, a fast decision path for transactions, and seamless fault tolerance—and demonstrated their inherent tensions with well-known multicore scalability properties. Specifically, we proved that serializable transactional systems that guarantee a minimal progress condition cannot satisfy all desirable properties, but could satisfy any subset of them. Future research could thus focus on designing practical algorithms that give up just one property, chosen according to the targeted applications and workloads. Examining whether the tension still exists between weaker variants of the properties we considered could also contribute to the future development of improved transactional systems.

Fault Tolerance

We studied fault tolerance in the presence of different kinds of failures across different systems. In Chapter 6 we considered systems utilizing new NVRAM technologies. We introduced a guideline for designing efficient durable algorithms for current NVRAM architectures, dictating to reduce accesses to flushed memory. We demonstrated the advantage of following this guideline with durable queues. After initially implementing queues that adhered to the known guideline of minimizing the fence count and not achieving the expected performance gains, we further amended them to ensure zero accesses to flushed memory. This led to a significant performance improvement on Intel Optane NVRAM, which highlights the importance of our general guideline. This guideline may contribute to an efficient design of future

implementations of concurrent durable data structures, including other queue implementations, a research topic that has garnered attention from others as well [e.g., FKK22; FGM24].

In Chapter 7 we examined another kind of fault: unlimited alteration channel noise in distributed systems. We demonstrated that content-oblivious computation is feasible in any 2-edge connected network, but a non-trivial content-oblivious computation is impossible in networks that are not 2-edge connected. This finding has applications in environments with severe noise, and ones with limited communication where nodes can only signal each other with pulses. Improving the overhead of content-oblivious simulations and analyzing the minimal possible overhead and its relationship with the network topology remains an open question. Tailoring content-oblivious computation for specific tasks could help reduce the overhead. Our work has already sparked further research aimed at eliminating the assumption of a preselected distinguished node, as evident in a study presenting a content-oblivious algorithm for leader election on rings [FGGN24].

Bibliography

[22] Java platform version 18 api specification. 2022. URL: https://docs.oracle.com/en/java/javase/18/docs/api/index.html.

[AAD+93] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *JACM*, 40(4), 1993.

[AB18] Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. In *PPoPP*, 2018.

[ABC+18] Marcos K Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pages 51–60, 2018.

[ABE+19] Noga Alon, Mark Braverman, Klim Efremenko, Ran Gelles, and Bernhard Haeupler. Reliable communication over highly connected noisy networks. *Distributed Computing*, 32(6):505–515, 2019. URL: https://doi.org/10.1007/s00446-017-0303-5.

[ABG+19] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of RDMA on agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 409–418, 2019.

[ABG+20] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616, 2020.

[ADG+21] Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. Leaderless consensus. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 392–402. IEEE, 2021.

[ADHS18] Abhinav Aggarwal, Varsha Dani, Thomas P. Hayes, and Jared Saia. Sending a message with unknown noise. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, ICDCN '18, Varanasi, India. Association for Computing Machinery, 2018.

[ADHS20] Abhinav Aggarwal, Varsha Dani, Thomas P. Hayes, and Jared Saia. A scalable algorithm for multiparty interactive communication with private channels. In *Proceedings of the 21st International Conference on Distributed Computing and Networking*, ICDCN 2020. Association for Computing Machinery, 2020.

[AF03] Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241*, 2003. URL: https://hpl.hp.com/techreports/2003/HPL-2003-241.html.

[AF15] Hagit Attiya and Panagiota Fatourou. Disjoint-access parallelism in software transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pages 72–97. Springer, 2015.

[AH12] Hagit Attiya and Eshcar Hillel. The cost of privatization in software transactional memory. *IEEE Transactions on Computers*, 62(12):2531–2543, 2012.

[AHM11] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computing Systems*, 49(4):698–719, 2011.

[AHS94] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *JACM*, 41(5), 1994.

[AS08] Hillel Avni and Nir Shavit. Maintaining consistent transactional states without a global clock. In *Colloquium on Structural Information & Communication Complexity*, 2008.

[AS10] Hiroyuki Akinaga and Hisashi Shima. Resistive random access memory (ReRAM) based on metal oxides. *Proceedings of the IEEE*, 98(12), 2010.

[AST12] Yehuda Afek, Nir Shavit, and Moran Tzafrir. Interrupting snapshots and the java™ size method. *Journal of Parallel and Distributed Computing*, 72(7), 2012.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*. John Wiley & Sons, 2004.

[BCP16] Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. CBPQ: high performance lock-free priority queue. In *Euro-Par*, 2016.

[BDFG14] Victor Bushkov, Dmytro Dziuma, Panagiota Fatourou, and Rachid Guerraoui. The pcl theorem: transactions cannot be parallel, consistent and live. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 178–187, 2014.

[BDM93] Michael Barborak, Anton Dahbura, and Miroslaw Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys (CSur)*, 25(2):171–220, 1993.

[BDY+21] Heng Bu, Ming-Kai Dong, Ji-Fei Yi, Bin-Yu Zang, and Hai-Bo Chen. Revisiting persistent indexing structures on intel optane dc persistent memory. *Journal of Computer Science and Technology*, 36(1), 2021.

[BEGH17] Mark Braverman, Klim Efremenko, Ran Gelles, and Bernhard Haeupler. Constant-rate coding for multiparty interactive communication is impossible. *J. ACM*, 65(1):4:1–4:41, December 2017.

[BGT15] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *OPODIS*, 2015.

[BHG86] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986.

[Bie03] Martin Biely. An optimal byzantine agreement algorithm with arbitrary node and link failures. In *Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems*, volume 1, pages 146–151, 2003.

[Boy14] Silas Boyd-Wickizer. *Optimizing communication bottlenecks in multiprocessor operating system kernels*. PhD thesis, Massachusetts Institute of Technology, 2014.

[BP12] Anastasia Braginsky and Erez Petrank. A lock-free B+ tree. In *SPAA*, 2012.

[BR21] Hadi Brais and Andy Rudoff. Reply to on x86-64, is the ”movnti” or ”movntdq” instruction atomic when system crash? 2021. URL: https://stackoverflow.com/a/65587308/7289606.

[Bro18] Trevor Brown. Java lock-free data structure library. 2018. URL: https://bitbucket.org/trbot86/implementations/src/master/java/src/algorithms/published.

[BSS23a] Naama Ben-David, Gal Sela, and Adriana Szekeres. The FIDS theorems: tensions between multinode and multicore performance in transactional systems. In *DISC*, 2023.

[BSS23b] Naama Ben-David, Gal Sela, and Adriana Szekeres. The FIDS theorems: tensions between multinode and multicore performance in transactional systems. *arXiv preprint*, 2023. eprint: 2308.03919.

[CBB14] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10), 2014.

[CCA+11] 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 *ASPLOS*, 2011.

[CCGS22] Keren Censor-Hillel, Shir Cohen, Ran Gelles, and Gal Sela. Distributed computations in fully-defective networks. In *PODC*, 2022.

[CCGS23] Keren Censor-Hillel, Shir Cohen, Ran Gelles, and Gal Sela. Distributed computations in fully-defective networks. *DC*, 2023.

[CDE+12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, 2012.

[CFL17] Nachshon Cohen, Michal Friedman, and James R Larus. Efficient logging in non-volatile memory by exploiting coherency protocols. *PACMPL*, 1(OOPSLA), 2017.

[CFR18] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: efficient algorithms for persistent transactional memory. In *SPAA*, 2018.

[CFR20] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *EuroSys*, 2020.

[CGH19] Keren Censor-Hillel, Ran Gelles, and Bernhard Haeupler. Making asynchronous distributed computations robust to noise. *Distributed Computing*, 32(5):405–421, October 2019.

[CGZ18] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *SPAA*, 2018.

[CJ15] Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *VLDB*, 8(7), 2015.

[CKZ+13] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, Farminton, Pennsylvania. Association for Computing Machinery, 2013.

[CL12] James Cowling and Barbara H. Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *Proceedings of 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, pages 223–236, 2012.

[CST+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[Das98] Pallab Dasgupta. Agreement under faulty interfaces. *Information Processing Letters*, 65(3):125–129, 1998.

[DDGZ18] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *USENIX ATC*, 2018. URL: https://usenix.org/conference/atc18/presentation/david.

[DGL05] Partha Dutta, Rachid Guerraoui, and Leslie Lamport. How fast can eventual synchrony lead to consensus? In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 22–27. IEEE, 2005.

[DGT15] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: the secret to scaling concurrent search data structures. In *ASPLOS*, 2015.

[DHW97] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *JACM*, 44(6), 1997.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[DMSY15] Varsha Dani, Mahnush Movahedi, Jared Saia, and Maxwell Young. Interactive communication with unknown noise rate. In *Automata, Languages, and Programming: 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, pages 575–587. Springer Berlin Heidelberg, 2015.

[DNN+15] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, Monterey, California. Association for Computing Machinery, 2015. URL: https://doi.org/10.1145/2815400.2815425.

[Dol82] Danny Dolev. The byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

[DTM+18] Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. Parallelization of dynamic languages: synchronizing built-in collections. *PACMPL*, 2(OOPSLA), 2018.

[Dub13] Elena Dubrova. *Fault-Tolerant Design*. Springer, 2013.

[EFRvB10] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *PODC*, 2010.

[EGG+22] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: a scalable, predictably performant, and fully managed NoSQL database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[EHK20] Klim Efremenko, Elad Haramaty, and Yael Tauman Kalai. Interactive Coding with Constant Round and Communication Blowup. In *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*, volume 151 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 7:1–7:34. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020.

[FBW+20] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. NVTraverse: in NVRAM data structures, the destination is more important than the journey. In *PLDI*, 2020.

[FFMR10] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

[FGGN24] Fabian Frei, Ran Gelles, Ahmed Ghazy, and Alexandre Nolin. Brief announcement: content-oblivious leader election on rings. In *PODC*, 2024.

[FGM24] Panagiota Fatourou, Nikos Giachoudis, and George Mallis. Highly-efficient persistent FIFO queues. In *SIROCCO*, 2024.

[FHMP18] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *PPoPP*, 2018.

[FHMV04] Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Vardi. *Reasoning about knowledge*. MIT press, 2004.

[FK12] Panagiota Fatourou and Nikolaos D Kallimanis. Revisiting the combining synchronization technique. In *PPoPP*, 2012.

[FKK22] Panagiota Fatourou, Nikolaos D Kallimanis, and Eleftherios Kosmas. The performance power of software combining in persistence. In *PPoPP*, 2022.

[FLP85] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2), 1985.

[FPR21] Michal Friedman, Erez Petrank, and Pedro Ramalhete. Mirror: making lock-free data structures persistent. In *PLDI*, 2021.

[FR24] Panagiota Fatourou and Eric Ruppert. Lock-free augmented trees. *arXiv preprint*, 2024. eprint: 2405.10506.

[Fra04] Keir Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, UK, 2004.

[FTA14] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD '14*, 2014.

[Gel17] Ran Gelles. Coding for interactive communication: a survey. *Foundations and Trends® in Theoretical Computer Science*, 13(1–2):1–157, 2017.

[GI20] Ran Gelles and Siddharth Iyer. Interactive Coding Resilient to an Unknown Number of Erasures. In *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153, 13:1–13:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020.

[GK08] Rachid Guerraoui and Michal Kapalka. On obstruction-free transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 304–313, 2008.

[GK19] Ran Gelles and Yael T. Kalai. Constant-rate interactive coding is impossible, even in constant-degree networks. *IEEE Transactions on Information Theory*, 65(6):3812–3829, 2019.

[GKR19] Ran Gelles, Yael Tauman Kalai, and Govind Ramnarayan. Efficient multiparty interactive coding for insertions, deletions, and substitutions. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 137–146, Toronto ON, Canada. ACM, 2019. URL: https://doi.org/10.1145/3293611.3331621.

[GL04] Rachid Guerraoui and Ron R Levy. Robust emulations of shared memory in a crash-recovery model. In *ICDCS*, 2004.

[GLR95] Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: observations and applications in tolerating hybrid and link faults. In *Dependable Computing and Fault Tolerant Systems*, volume 10, pages 139–158. IEEE Computer Society, 1995. URL: http://www.csl.sri.com/papers/dcca95/dcca95.pdf.

[GMS14] Ran Gelles, Ankur Moitra, and Amit Sahai. Efficient coding for interactive communication. *IEEE Transactions on Information Theory*, 60(3):1899–1913, March 2014.

[Gra78] J.N. Gray. Notes on data base operating systems. In *Operating Systems*. Volume 60, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 1978.

[Hao19] Xiangpeng Hao. Is clwb actually implemented? 2019. URL: https://blog.haoxp.xyz/posts/is-clwb-implemented.

[Har01] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, 2001.

[Her91] Maurice Herlihy. Wait-free synchronization. *TOPLAS*, 13(1), 1991.

[HHL+05] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, 2005.

[HLH+13] Andreas Haas, Michael Lippautz, Thomas A Henzinger, Hannes Payer, Ana Sokolova, Christoph M Kirsch, and Ali Sezgin. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *CF*, 2013.

[HLLS07] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *SIROCCO*, 2007.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, 2003.

[HM90] Joseph Y Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM (JACM)*, 37(3):549–587, 1990.

[HP21a] Yael Hitron and Merav Parter. Broadcast CONGEST algorithms against adversarial edges. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, 23:1–23:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: https://doi.org/10.4230/LIPIcs.DISC.2021.23.

[HP21b] Yael Hitron and Merav Parter. General CONGEST compilers against adversarial edges. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, 24:1–24:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: https://doi.org/10.4230/LIPIcs.DISC.2021.24.

[HS08] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[HS16] William M. Hoza and Leonard J. Schulman. The adversarial noise threshold for distributed protocols. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, chapter 18, pages 240–258, 2016.

[HSS07] Moshe Hoffman, Ori Shalev, and Nir Shavit. The baskets queue. In *OPODIS*, 2007.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.

[IBM] IBM. IBM MQ. URL: https://ibm.com/software/products/en/ibm-mq.

[IMS16] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *DISC*, 2016.

[Int19] Intel. 3D XPoint™ : a breakthrough in non-volatile memory technology. 2019. URL: https://intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[Int20] Intel. Intel®64 and IA-32 architectures software developer's manual. 2020. URL: https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf.

[IR94] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 151–160, Los Angeles, California, USA. Association for Computing Machinery, 1994. URL: https://doi.org/10.1145/197917.198079.

[Jay05] Prasad Jayanti. An optimal multi-writer snapshot algorithm. In *STOC*, 2005.

[JKL15] Abhishek Jain, Yael Tauman Kalai, and Allison Lewko. Interactive coding for multiparty protocols. In *Proceedings of the 6th Conference on Innovations in Theoretical Computer Science, ITCS '15*, pages 1–10, 2015.

[KAA24] Ilya Kokorin, Dan Alistarh, and Vitaly Aksenov. Wait-free trees supporting asymptotically efficient range queries. In *IPDPS*, 2024.

[KAK20] Anuj Kalia, David Andersen, and Michael Kaminsky. Challenges and solutions for fast remote persistent memory access. In *SoCC*, 2020.

[KK20] Israel Koren and C. Mani Krishna, editors. *Fault-Tolerant Systems*. Morgan Kaufmann, San Francisco (CA), second edition edition, 2020, page xi.

[KLP13] Christoph M Kirsch, Michael Lippautz, and Hannes Payer. Fast and scalable, lock-free k-FIFO queues. In *PaCT*, 2013.

[KPF+13] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, Prague, Czech Republic. Association for Computing Machinery, 2013. URL: https://doi.org/10.1145/2465351.2465363.

[KPS+16] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. In *ASPLOS*, 2016.

[KR00] A. Kazmierczak and S. Radhakrishnan. An optimal distributed ear decomposition algorithm with applications to biconnectivity and outerplanarity testing. *IEEE Transactions on Parallel and Distributed Systems*, 11(2):110–118, 2000.

[KR01] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: preliminary version. *ACM SIGACT News*, 32(2):45–63, 2001.

[KWQ+12] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 237–250, Hollywood, CA, USA. USENIX Association, 2012.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*:51–58, 2001.

[Lam06a] Leslie Lamport. Fast paxos. *Distributed Comput.*, 19(2):79–103, 2006. URL: https://doi.org/10.1007/s00446-006-0005-x.

[Lam06b] Leslie Lamport. Lower bounds for asynchronous consensus. *Distrib. Comput.*, 19(2):104–125, October 2006. URL: https://doi.org/10.1007/s00446-006-0155-x.

[Lam06c] Leslie Lamport. Lower bounds for asynchronous consensus. *Distrib. Comput.*, 19(2):104–125, October 2006. URL: https://doi.org/10.1007/s00446-006-0155-x.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. URL: https://doi.org/10.1145/359545.359563.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[Lea04] Doug Lea. The java concurrency package (jsr-166), 2004. URL: http://gee.cs.oswego.edu/dl/concurrency-interest.

[Lea09] Doug Lea. The java concurrency package (JSR-166), 2009.

[LLS+17] 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 *FAST*, 2017.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010. URL: https://doi.org/10.1145/1773912.1773922.

[LMP17] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, SOSP '17, 2017.

[Lov85] L. Lovasz. Computing ears and branchings in parallel. In *26th Annual Symposium on Foundations of Computer Science*, pages 464–467, 1985.

[LS04] Edya Ladan-Mozes and Nir Shavit. An optimistic approach to lock-free FIFO queues. In *Distributed Computing*, 2004.

[LV15] Allison Lewko and Ellen Vitercik. Balancing communication for multi-party interactive coding, 2015. arXiv: 1503.06381.

[Lyn96] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[MA13] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *PPoPP*, 2013.

[MAK12] Iulian Moraru, David G Andersen, and Michael Kaminsky. Egalitarian paxos. In *ACM Symposium on Operating Systems Principles*, 2012.

[Mic02] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, 2002.

[MIS20] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: easy and fast persistence for volatile data structures. In *ASPLOS*, 2020.

[MMT+18] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, et al. Persistent memory transactions. *arXiv preprint*, 2018. eprint: 1804.00701.

[MNLL16] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 517–532, Savannah, GA, USA. USENIX Association, 2016.

[MRG16] Remigius Meier, Armin Rigo, and Thomas R Gross. Parallel virtual machines with rpython. In *DLS*, 2016.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[NHP22] Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. Bundling linked data structures for linearizable range queries. In *PPoPP*, 2022.

[NM14] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *PPoPP*, 2014.

[OLN+16] 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 *SIGMOD*, 2016.

[OM20] Or Ostrovsky and Adam Morrison. Scaling concurrent queues by using HTM to profit from failed atomic operations. In *PPoPP*, 2020.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014.

[Ora] Oracle. Oracle Tuxedo Message Queue. URL: https://docs.oracle.com/cd/E35855_01/otmq/docs12c/overview/overview.html.

[Pap79] Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[Pel00] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000.

[Pel92] Andrzej Pelc. Reliable communication in networks with byzantine link failures. *Networks*, 22(5):441–459, 1992.

[PPR+15] Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Binoy Ravindran, and Francesco Quaglia. Disjoint-access parallelism: impossibility, possibility, and cost of transactional memory implementations. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 217–226, 2015.

[PT13] Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *DISC*, 2013.

[PT86] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, SE-12(3):477–482, 1986.

[PY19a] Merav Parter and Eylon Yogev. Low congestion cycle covers and their applications. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1673–1692. SIAM, 2019. URL: https://doi.org/10.1137/1.9781611975482.101.

[PY19b] Merav Parter and Eylon Yogev. Optimal Short Cycle Decomposition in Almost Linear Time. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132, 89:1–89:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[Ram93] Vijaya Ramachandran. Parallel open ear decomposition applications to graph biconnectivity and triconnectivity. In *Synthesis of Parallel Algorithms*, chapter 7. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[Ray18] Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems: An Algorithmic Approach*. Springer, Cham, 2018, page 459.

[RBB+08] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: a scalable technology. *IBM Journal of Research and Development*, 52(4.5), 2008.

[RCFC19] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. OneFile: a wait-free persistent transactional memory. In *DSN*, 2019.

[RHH09] Amitabha Roy, Steven Hand, and Tim Harris. Exploring the limits of disjoint access parallelism. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism, Berkeley, CA*, 2009.

[Rob39] Herbert Ellis Robbins. A theorem on graphs, with an application to a problem of traffic control. *The American Mathematical Monthly*, 1939.

[RS94] Sridhar Rajagopalan and Leonard Schulman. A coding theorem for distributed computation. In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 790–799, 1994.

[RST01] Yaron Riany, Nir Shavit, and Dan Touitou. Towards a practical snapshot algorithm. *Theoretical Computer Science*, 269(1-2), 2001.

[Rud19] Andy Rudoff. Reply to how to use clwb instructions. 2019. URL: https://groups.google.com/g/pmem/c/R8H3sKq9sLQ/m/ltL7Kng4BAAJ.

[Rud20] Andy Rudoff. Reply to 8 byte atomicity & larger store operations. 2020. URL: https://groups.google.com/g/pmem/c/6%5C_5daOuEI00/m/nY%5C_mtKd0CAAJ.

[RWNV20] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-x86 architecture. *PACMPL*, 4(POPL), 2020.

[SAA95] Hasan M. Sayeed, Marwan Abu-Amara, and Hosame Abu-Amara. Optimal asynchronous agreement and leader election algorithm for complete networks with byzantine faulty links. *Distributed Computing*, 9(3):147–156, December 1995.

[SAKM09] Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. Rpc chains: efficient client-server communication in geodistributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, 2009.

[Sca20] Steve Scargall. *Programming persistent memory: A comprehensive guide for developers*. Springer Nature, 2020.

[Sch13] Jens M. Schmidt. A simple test on 2-vertex- and 2-edge-connectivity. *Information Processing Letters*, 113(7):241–244, 2013.

[Sch92] Leonard J. Schulman. Communication on noisy channels: a coding theorem for computation. *Foundations of Computer Science, Annual IEEE Symposium on*:724–733, 1992.

[Sch93] Leonard J. Schulman. Deterministic coding for interactive communication. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 747–756, San Diego, California, United States. ACM, 1993.

[SCY98] Hin-Sing Siu, Yeh-Hao Chin, and Wei-Pang Yang. Byzantine agreement in the presence of mixed faults on processors and links. *IEEE Transactions on Parallel and Distributed Systems*, 9(4):335–345, April 1998.

[SGP18] Gali Sheffi, Guy Golan-Gueta, and Erez Petrank. A scalable linearizable multi-index table. In *ICDCS*, 2018.

[SHP21a] Gal Sela, Maurice Herlihy, and Erez Petrank. Brief announcement: linearizability: a typo. In *PODC*, 2021.

[SHP21b] Gal Sela, Maurice Herlihy, and Erez Petrank. Linearizability: a typo. *arXiv preprint*, 2021. eprint: 2105.06737.

[SKL+18] Gal Sela-Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. BQ: a lock-free queue with batching. In *SPAA*, 2018.

[SNI17] SNIA. NVM programming model (NPM). 2017. URL: https://snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf.

[Sof] Pivotal Software. RabbitMQ. URL: https://rabbitmq.com.

[SP21a] Gal Sela and Erez Petrank. Durable queues: the second amendment. In *SPAA*, 2021.

[SP21b] Gal Sela and Erez Petrank. Durable queues: the second amendment. *arXiv preprint*, 2021. eprint: 2105.08706.

[SP22a] Gal Sela and Erez Petrank. Concurrent size. *arXiv preprint*, 2022. eprint: 2209.07100.

[SP22b] Gal Sela and Erez Petrank. Concurrent size. *PACMPL*, 6(OOPSLA2), 2022.

[SP22c] Gal Sela and Erez Petrank. Concurrent size - artifact for oopsla'22. 2022.

[SP24] Gal Sela and Erez Petrank. Concurrent aggregate queries. *arXiv preprint*, 2024. eprint: 2405.07434.

[SRN+19] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *SIGMOD'19*, 2019.

[SS05] William N Scherer III and Michael L Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, 2005.

[ST05] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5), 2005.

[Sto85] Michael Stonebraker. The case for shared nothing. In *IEEE Database Eng. Bull.* 1985.

[SW90] Nicola Santoro and Peter Widmayer. Distributed function evaluation in the presence of transmission faults. In *International Symposium on Algorithms*, pages 358–367. Springer, 1990.

[SWK09] Ulrich Schmid, Bettina Weiss, and Idit Keidar. Impossibility results and lower bounds for consensus under link failures. *SIAM Journal on Computing*, 38(5):1912–1951, 2009.

[SWL+20] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. Meerkat: multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[SZ96] Nir Shavit and Asaph Zemach. Diffracting trees. *TOCS*, 14(4), 1996.

[tea] Intel PMDK team. Persistent memory programming. URL: https://pmem.io.

[Tsi04] Y.H Tsin. On finding an ear decomposition of an undirected graph distributively. *Information Processing Letters*, 91(3):147–153, 2004.

[TZK+13a] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, Farminton, Pennsylvania. Association for Computing Machinery, 2013. URL: https://doi.org/10.1145/2517349.2522713.

[TZK+13b] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[vRVL+19] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory I/O primitives. In *DaMoN*, 2019.

[VS04] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[VTS11] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: lightweight persistent memory. In *ASPLOS*, 2011.

[WBB+21] Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *PPoPP*, 2021.

[WCD+20] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L Scott. Montage: a general system for buffered durably linearizable data structures. *arXiv preprint*, 2020. eprint: 2009.13701.

[Wei21] Yuanhao Wei. Vcaslib. 2021. URL: https://github.com/yuanhaow/vcaslib.

[Whi32] Hassler Whitney. Non-separable and planar graphs. *Transactions of the American Mathematical Society*, 34(2):339–362, 1932. URL: http://www.jstor.org/stable/1989545.

[WJC+17] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 94–107, 2017.

[WRL19] Kai Wu, Jie Ren, and Dong Li. Architecture-aware, high performance transaction for persistent memory. *arXiv preprint*, 2019. eprint: 1903.06226.

[YKH+20] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *FAST*, 2020.

[YM16] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *PPoPP*, 2016.

[YMR+19] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[YPSD16] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*, 2016.

[YWC+15] 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 *FAST*, 2015.

[ZFS+19] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *PACMPL*, 3(OOPSLA), 2019.

[ZSS+15] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 263–278, Monterey, California. Association for Computing Machinery, 2015. URL: https://doi.org/10.1145/2815400.2815404.

[ZXS+21] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. Foundationdb: a distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD'21)*, 2021.

[ZZLS19] Pantea Zardoshti, Tingzhe Zhou, Yujie Liu, and Michael Spear. Optimizing persistent memory transactions. In *PACT*, 2019.

מוליכים למחצה חדשות שפותחו על ידי אינטל וחברות אחרות. מדובר בתורים עמידים לנפילות - שיכולים להמשיך לתפקד עם עלית המחשב מנפילת חשמל. תורים אלו לא רק מציגים ביצועים גבוהים, אלא גם מוכחים כאופטימליים במזעור האינטראקציה עם הזיכרון, ועומדים בהנחיה כללית חדשה שאנו מציגים לעיצוב יעיל של אלגוריתמים לזיכרון לא נדיף. בעת ניתוח נכונות של מבני נתונים מקביליים, הלינאריזביליות היא הקריטריון בשימוש הנפוץ ביותר. תוך כדי התעמקות בתנאי זה, מצאנו טעות בהגדרה שהוצגה במאמר המקורי ואנו מספקים תיקון להפיכת ההגדרה לשלמה.

תחת מערכות מבוזרות, אנו מתמקדים בעמידות בפני תקלות, אתגר יסודי בתכנון מערכות מבוזרות. אנו בוחנים רשתות אסינכרוניות עם רעש בלתי מוגבל בערוצים, שעלול לשבש את כל ההודעות בכל הערוצים, ועל כן דורש תקשורת בלתי תלויה בתוכן. אנו מראים כיצד רשתות 2-קשירות-קשתות מסוגלות באופן מפתיע להתמודד עם רעש בלתי מוגבל בערוצים, בעוד שרשתות אחרות אינן מסוגלות. יתר על כן, אנו בוחנים מערכות מבוזרות המשלבות מקביליות, כלומר, כוללות מספר צמתים כאשר כל צומת הוא מכונה מרובת ליבות. אנו מנסחים תכונות קריטיות למהירות ועמידות של מערכות מבוזרות, ומציגים שקלול תמורות ביניהן לבין תכונות רצויות ליעילות במערכת מרובת ליבות. הדבר עשוי לשמש כקו מנחה בעיצוב מערכות משולבות עתידיות עם ביצועים גבוהים.

תקציר

מאז המצאת המחשב, אחת המטרות העיקריות של תעשית המחשבים היתה לשפר את ביצועי המחשב, ולהפכו לכלי חיוני בחברה המודרנית. בתחילה, מחשבים הכילו ליבה אחת בה התוכניות הורצו, ושדרוגי חומרה מתמשכים של הליבה הובילו למחשבים חדשים שהריצו את אותה תוכנה מהר יותר. עם זאת, החל מתחילת שנות ה-2000, לא התאפשר עוד לבצע שיפורים נוספים באותה שיטה עקב מגבלות פיזיות (בעיקר התחממות יתר). כתחליף, דרך מרכזית לשיפור ביצועי מעבדי המחשב בעשרים השנים הינה ייצור מעבדים מרובי ליבות. מעבדים אלו מאפשרים הרצה מקבילה של תהליכים רבים ובכך משפרים את הביצועים. עם זאת, כדי לנצל חומרה זו, נדרשים אלגוריתמים חדשים לסנכרון הגישה לנתונים בין הליבות השונות. תכנון אלגוריתמים מקביליים מציב אתגר משמעותי כיום במדעי המחשב. אבני יסוד בסיסיות של אלגוריתמים אלו, שתזה זו מתמקדת בהן, הן מבני נתונים מקביליים.

נושא אחר שעומד במרכז תזה זו הוא מערכות מבוזרות. בעולם המקושר שלנו, יישומים מודרניים דורשים לעתים קרובות כח מחשוב ועמידות לתקלות שמחשב יחיד אינו יכול להציע. על כן תשתית המחשוב המודרנית מסתמכת על מערכות מבוזרות, המאגדות מספר מחשבים שמתקשרים באמצעות העברת הודעות בערוצי תקשורת לשם ביצוע משימות משותפות. שילוב מספר מחשבים משפר את היעילות הכוללת של המערכת ומבטיח שירות רציף ללקוחות, גם במקרים של תקלות שרתים או הפרעות ברשת.

במחקר של מערכות מרובות תהליכים, בין אם מקביליות, מבוזרות, או שילוב של שתיהן (מערכות עם מספר מחשבים שכל אחד מהם מריץ מספר תהליכים), יש לתת את הדעת על סנכרון ושיתוף פעולה נכונים ויעילים של התהליכים להשגת מטרתם המשותפת. ביצועים גבוהים אינם הגורם היחיד שיש לקחת בחשבון כשמתכננים מערכות מקביליות או מבוזרות, אלא יש להבטיח גם נכונות ועמידות בפני תקלות. אנו חוקרים אלגוריתמים מקביליים ומבוזרים יסודיים המספקים תכונות חיוניות אלו. בתזה זו אנו חוקרים את גבולות החישוב כדי לקבוע מה ניתן ומה לא ניתן להשיג במערכות מרובות תהליכים. התזה מתארת את מרחב הפתרונות האפשריים לבעיות שונות, מאפיינת את מה שלא ניתן לפתור, ומנסחת קווים מנחים בתוך התחום האפשרי ליישומים יעילים בעתיד. בנוסף, אנו דוחפים את גבולות האפשר ומציגים אלגוריתמים חדשים המשפרים את הספרות הקיימת, כפי שאנו מוכיחים הן תיאורטית והן על ידי מדידות.

בתחום מבני הנתונים המקביליים, אנו בוחנים את תכונת הגודל שלהם ומציגים את השיטה הראשונה להוספת פעולה שמחזירה את גודל המבנה באופן מהיר ונכון, שיטה שניתן להחיל על קבוצה רחבה של מבני נתונים מקביליים. ניתן להפעיל את שיטתנו על מבני נתונים מקביליים בספריות בשפות ג'אווה, סי פלאס פלאס, פייתון ועוד. אנו מתמודדים גם עם המשימה של עיצוב תורי נכנס-ראשון-יוצא-ראשון מקביליים יעילים עבור מחשבים עם זיכרון לא נדיף, סוג חדש של זיכרון המשתמש בטכנולוגיות

המחקר בוצע בהנחייתו של פרופ' ארז פטרנק, בפקולטה למדעי המחשב בטכניון.

מחברת חיבור זה מצהירה כי המחקר, כולל איסוף הנתונים, עיבודם והצגתם, התייחסות והשוואה למחקרים קודמים וכו', נעשה כולו בצורה ישרה, כמצופה ממחקר מדעי המבוצע לפי אמות המידה האתיות של העולם האקדמי. כמו כן, הדיווח על המחקר ותוצאותיו בחיבור זה נעשה בצורה ישרה ומלאה, לפי אותן אמות מידה.

התוצאות בחיבור זה פורסמו כמאמרים מאת המחברת ושותפיה למחקר בכנסים ובכתבי-עת במהלך תקופת מחקר הדוקטורט של המחברת:

Gal Sela and Erez Petrank. Concurrent size. *PACMPL*, 6(OOPSLA2), 2022.

Gal Sela, Maurice Herlihy, and Erez Petrank. Brief announcement: linearizability: a typo. In *PODC*, 2021.

Naama Ben-David, Gal Sela, and Adriana Szekeres. The FIDS theorems: tensions between multinode and multicore performance in transactional systems. In *DISC*, 2023.

Gal Sela and Erez Petrank. Durable queues: the second amendment. In *SPAA*, 2021.

Keren Censor-Hillel, Shir Cohen, Ran Gelles, and Gal Sela. Distributed computations in fully-defective networks. In *PODC*, 2022.

Keren Censor-Hillel, Shir Cohen, Ran Gelles, and Gal Sela. Distributed computations in fully-defective networks. *DC*, 2023.

תודות

לא אוכל להודות מספיק למנחה שלי, פרופ' ארז פטרנק. ארז הוא אחד האנשים הנחמדים שיצא לי להכיר, כפי שיעיד כל מי שפגש אותו, וכל פגישה עם ארז היא לא רק מלמדת אלא גם מהנה. נהניתי ללמוד ממנו על מקביליות ולעבוד יחד על בעיות מרתקות. ארז הדריך אותי אקדמית ומקצועית, ומעבר לידע שצברתי מהעבודה איתו והתעצבותי כחוקרת בעבודתנו המשותפת, הוא גם הכיר לי לא מעט אנשים מהתחום. אבל ארז היה הרבה מעבר למנחה, ודאג לי גם בחיים האישיים, התחשב בצרכיי וסייע לי מעל ומעבר לכל מה שאפשר היה לבקש. זה עוד לפני שהזכרנו את הטיול המשותף עם ארז ויעל אחרי הכנס בניו זילנד, ואת הכנס שהמזוודה שלי לא הגיעה אליו ובלי ארז לא היתה לי אפילו פיג'מה...

אני מודה לשותפיי למחקר, קרן צנזור-הלל, רן גלס, שיר כהן, נעמה בן-דוד, אדריאנה צקרס, מוריס הרליהי, סמואל תומס, טלי מורשת, איריס בהר, אלכס קוגן, יוסי לב, ויקטור לוצ'נגקו וחן קאס-שריר על שעות של פגישות מרתקות ומעשירות ועבודה פוריה משותפת. במיוחד אני רוצה להודות לנעמה, שאירחה אותי להתמחות ב-VMWare. נהניתי מכל פגישה שלנו בקמפוס, אבל מעבר לכך, היא הזמינה אותי לדירתה הנעימה, ובעיקר - גרמה לי להרגיש בבית בתקופת ההתמחות.

אני מודה לבוחנים בועדות בחינות המועמדות והגמר שלי, יהודה אפק, קרן צנזור-הלל, רועי פרידמן, מוריס הרליהי ועדית קידר, על הזמן שלהם והתובנות שהם נתנו לי.

ולסיום תודה לאור אהובי ולמשפחתי שאין כמוה.

אני מודה לטכניון, למלגת ג'ייקובס, למלגת קנת' וגלוריה לוי ולקרן הלאומית למדע על התמיכה הכספית הנדיבה בהשתלמותי.

נכונות, יעילות ושרידות
של מערכות מקביליות ומבוזרות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

גל סלע

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
תמוז ה'תשפ"ד חיפה יולי 2024

נכונות, יעילות ושרידות
של מערכות מקביליות ומבוזרות

גל סלע