Theoretical Foundations for Practical Concurrent and Distributed Computation

Naama Ben-David

CMU-CS-20-126

August 26, 2020

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy Blelloch, Chair
Umut Acar
Phillip Gibbons
Mor Harchol-Balter
Erez Petrank (Technion)

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

This research was sponsored by the National Science Foundation award numbers: CCF1919223, CCF1533858, and CCF1629444; by a Microsoft Research PhD Fellowship; and by a National Sciences and Engineering Research Council of Canada (NSERC) Postgraduate Scholarship-Doctoral (PGS-D) fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Multiprocessor hardware, NUMA, contention, shared memory, NVRAM, RDMA

Abstract

Many large-scale computations are nowadays computed using several processes, whether on a single multi-core machine, or distributed over many machines. This wide-spread use of concurrent and distributed technology is also driving innovations in their underlying hardware. To design fast and correct algorithms in such settings, it is important to develop a theory of concurrent and distributed computing that is faithful to practice. Unfortunately, it is difficult to abstract practical problems into approachable theoretical ones, leading to theoretical models that are too far removed from reality to be easily applied in practice.

This thesis aims to bridge this gap by building a strong theoretical foundation for practical concurrent and distributed computing. The thesis takes a two-pronged approach toward this goal; improving theoretical analysis of well-studied settings, and modeling new technologies theoretically.

In the first vein, we consider the problem of analyzing shared-memory concurrent algorithms such that the analysis reflects their performance in practice. A new shared memory model that accounts for contention costs is presented, as well as a tool for uncovering the way that a machine interleaves concurrent instructions. We also show that the analysis of concurrent algorithms in more restricted settings can be easy and accurate, by designing and analyzing a concurrent algorithm for nested parallelism.

The second approach explored in this thesis to bridge the theory-practice gap considers and models two emerging technologies. Firstly, we study non-volatile random access memories (NVRAM) and develop a general simulation that can adapt many classic concurrent algorithms to a setting in which memory is non-volatile and can recover after a system fault. Finally, we study remote direct memory access (RDMA) as a tool for replication in data centers. We develop a model that captures the power of RDMA and demonstrate that RDMA supports heightened performance and fault tolerance, thereby uncovering its practical relevance by formally reasoning about it theoretically.

Acknowledgements

I would like to express my deep gratitude to my incredible advisor, Guy Blelloch. Being a good researcher and a good advisor are two very different skill-sets, and I feel extremely fortunate to have worked with someone who is so profoundly great at both. I've had the privilege of working closely with Guy over the five years of my PhD, allowing me to learn from his wide breadth of technical knowledge and his critical way of choosing worthy problems. At the same time, Guy has always allowed me to explore my own research interests, supporting and encouraging my various internships, visits, and other collaborations. I cannot think of a better role model, and I can only hope to continue to learn from him in the future.

I'd also like to thank my other thesis committee members, Umut Acar, Phil Gibbons, Mor Harchol-Balter, and Erez Petrank, for their time, their wisdom, and their advice. I especially thank Erez for his friendship, relentless support, and complete confidence in my ability as a researcher. He has made the Technion feel like a second home-base for me, welcoming me with office space, regular research meetings, and frequent lunch excursions whenever I visit.

During my PhD, I've had the opportunity to work with many amazing researchers, who have inspired me and made my research more enjoyable. I therefore thank my collaborators, Umut Acar, Marcos Aguilera, Guy Blelloch, Irina Calciu, David Yu Cheng Chan, Jeremy Fineman, Michal Friedman, Phil Gibbons, Yan Gu, Rachid Guerraoui, Vassos Hadzilacos, Virendra Marathe, Charles McGuffey, Erez Petrank, Mike Rainey, Ziv Scully, Julian Shun, Yihan Sun, Sam Toueg, Yuanhao (Hao) Wei, Athanasios (Thanasis) Xygkis, and Igor Zablotchi. I've especially enjoyed my collaborations with Hao; I thank him for the many insightful conversations, brainstorming sessions, and long deadline-fueled nights.

I would like to thank my internship/visit hosts, Marcos Aguilera and Rachid Guerraoui. Marcos hosted me during my internship at VMware, and introduced me to an exciting research area and to many opportunities that have enriched my PhD experience, even long after my internship ended. Throughout it all, Marcos has been a great collaborator and a reliable source of support and advice. Rachid invited me to visit him at EPFL, which became one of the most fun and productive semesters of my degree. I thank him for this opportunity, for his encouragement and for the examples he sets, both in research and in life.

The years spent working on my PhD have been the best in my life. This would not have been possible without my incredible friends. While there are too many friends to name here, I am grateful for each office chat, coffee break, walk, talk, lunch, dinner, karaoke night, library and coffee shop work session, work party, and real party. You have made Pittsburgh my home; the place I feel most comfortable and miss when I am away. I'd like to especially thank Bailey Flanigan and Ram Raghunathan. Bailey has kept me sane during the quarantine, providing com-

pany, laughter, support, good food, and overall close friendship whenever I needed it. Ram is an extraordinary friend whom I feel lucky to have in my life. I am looking forward to finally living in the same metropolitan area as him again!

My fiance, David Wajc, is both a family member and a friend. He has been there for me through the day-to-day frustrations and joys, making my successes more rewarding and my failures less discouraging. I thank him for the inexpressible comfort in the knowledge that despite leaving Pittsburgh behind, I will feel at home anywhere with him. Finally, I thank my mother Shoham Ben-David, my father Shai Ben-David, my two brothers Shalev and Sheffi Ben-David, and my sister Lior Hochman, for their unconditional love and support. They provided encouragement to pursue my passion, confidence that I can achieve anything, and the reassurance that research is not the most important thing in life. I dedicate this thesis to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As Moore's law comes to an end, the continued advancement of computation increasingly relies on the use of multiple processes. Nowadays, most phones and laptops have multiple CPUs, and many large-scale computations are computed using several machines. We depend on multiprocess computation for the speed, reliability, and scalability we have grown to expect from our day-to-day interactions with computers.

Multiprocess settings can be broadly divided into two categories; *concurrent* systems, which handle the interaction of multiple processes on the same machine, and *distributed* systems, which deal with communication among several machines. These two settings are generally modeled differently, according to how processes communicate with each other; the *shared-memory model* handles concurrent settings in which processes communicate by writing and reading from the same memory, and the *message-passing model* is considered for settings in which processes send each other messages, usually when distributed across separate machines. In either case, however, many of the same challenges arise; different processes often operate at different speeds, with the possibility of experiencing independent failures. Therefore the communication between processes often lacks full information, as it is difficult to distinguish between a process that is slow and a process that has experienced a fatal crash, and total order among the operations of different processes is impossible to establish. Despite these challenges, it is of the utmost importance to understand how to best use the multiprocess technology available to us, to continue to improve computations that rely on concurrent and distributed settings.

To achieve such an understanding, concurrent and distributed computing have been studied both in theory and in practice for decades. On the theory side, clean models have been developed to abstract away messy details and help reason about multiprocess executions [76, 110, 116, 119, 131, 150, 155]. Many efficient algorithms, as well as impossibility results, have been established with the help of this theoretical reasoning [29, 51, 77, 149]. On the practical side, decades of studies, work, and expertise have yielded highly efficient libraries of concurrent data structures [144, 199, 231, 232]. In fact, such results have not been achieved in isolation, and the field has greatly benefited from the mixing of theoretical and practical ideas. However, significant gaps between theory and practice are still left unaddressed.

Key features are lost in the abstraction of real hardware into a theoretical model, and theoretical results can therefore often miss crucial practical considerations. For example, existing theoretical models have difficulty capturing the running time of concurrent algorithms, and do

not distinguish between different types of hardware that can greatly increase the space of possible algorithmic designs. Furthermore, lack of a theoretical understanding of new and developing hardware features has led to missed opportunities in system design [13, 14].

In this thesis, we aim to bridge the gap between theory and practice in concurrent and distributed computing. Broadly speaking, we take two approaches to achieving this goal. Firstly, we study classic concurrent architectures that have already been modeled in the literature for decades, and refine the known model to allow for more accurate analysis of concurrent algorithms. A known drawback of the classic shared-memory model is that, while it allows developing robust algorithms that are provably correct under any possible scenario, analyzing the running time of these algorithms is extremely difficult. In fact, most works, after rigorously proving an algorithm correct, evaluate its efficiency through empirical analysis alone [111, 112, 144, 235, 274]. It has been noted that relying so heavily on empirical analysis can lead to false conclusions or to missed problematic scenarios unless done with extreme care [145]. We therefore aim to facilitate the theoretical analysis of running time for shared-memory algorithms. We take a few approaches toward this goal; we first present a model that separates different causes of asynchrony in the system, and we show that this model allows analyzing a protocol known as exponential backoff, which was never analyzed in shared memory before. Secondly, we develop a tool for viewing the behavior of concurrent executions on modern machines, thereby allowing us to study these executions and learn important patterns. Finally, we show that analysis is sometimes possible using the classic model if we consider the context in which an algorithm is run. In particular, we show that in a nested parallel computation, which imposes some structure on the concurrent execution, it is possible to design and analyze provably efficient algorithms.

To be able to have theory that is meaningful in practice, it is important to model the real capabilities of the hardware. However, this can be surprisingly challenging, since hardware is ever changing and developing. Some technological advancements not only improve the speed of processing power and memory, but fundamentally change the system behavior, or introduce new possibilities altogether. Failing to model useful features of hardware can result in researchers overlooking opportunities that these technologies can offer.

Our second approach to bridging the theory-practice gap therefore considers modeling emerging hardware that has features that were not accounted for in classic theoretical work. We focus on two such technologies; non-volatile random access memory (NVRAM), and remote direct memory access (RDMA).

NVRAM is a new memory technology that provides persistence at speeds comparable to DRAM. It opens the potential to design persistent data structures that are much faster than the ones that exist today, which normally rely on much slower disk technology. However, NVRAM also opens up new challenges, since it cannot make caches persistent, meaning that simply running a concurrent algorithm in a system with NVRAM would lose any data stored on cache upon a system fault. Care must be taken in designing algorithms that store enough data on main memory to be able to recover after such a fault despite losing some data. NVRAM has been heavily studied in recent years [44, 56, 60, 80, 89, 92, 136, 166, 201, 202]. We discuss models for NVRAM [30, 59, 166], and present the first general simulation result that can convert any concurrent algorithm designed in the classic shared-memory model into an algorithm that uses NVRAM and can recover after a fault.

RDMA is a communication technology among machines in a data center that allows different

machines to access each other's memory without involving the host CPU. This capability is akin to supporting shared-memory accesses among processes that normally communicate by sending messages over a network. Communication over RDMA is also significantly faster than TCP/IP. Given these features, RDMA-based systems have received considerable attention in recent years, with fast RDMA-based distributed key-value stores and state machine replication systems being designed [41, 106, 107, 170, 172, 173, 183, 250, 279]. We present the first theoretical model that captures RDMA's capabilities, and prove that it can enhance the fault tolerance of systems that use it, as well as reduce the number of network round trips of communication required to solve agreement problems among processes. We present several results that focus on scalability, fault tolerance, and performance of agreement tasks. Finally, we use some of the ideas from the theoretical work to build a state machine replication system that is faster than all previously known systems, both in the common case and when handling failures. This therefore supports the message of this thesis, that theoretical understanding of a problem can lead to better solutions in practice.

The thesis is organized into three parts. The first presents new ways of analyzing concurrent algorithms on classic hardware, the second studies NVRAM, and the third presents our contributions on modeling and using RDMA. In the remainder of this chapter, we briefly summarize the results of each part, and then discuss some preliminaries that are needed to understand the technical contributions presented in the thesis.

1.1 Part 1: Contention in Shared Memory

The behavior of concurrent shared memory is difficult to predict. There are many factors that may affect the performance of a given concurrent algorithm; from the number of processors available, to the structure of the memory hierarchy, and even to how busy the machine is. It is fair to assume that every time a concurrent algorithm runs, its behavior will be different. To reason about such a chaotic setting, theoreticians often abstract away these concerns by considering an omnipotent adversarial scheduler that determines the order of events in the system. A bound or a proof of correctness must then hold under any schedule that the adversary chooses. In this way, we can guarantee that any real execution will be at least as good as the theoretical bounds.

However, as it turns out, the theoretical guarantees that can be derived under such an adversarial model are too pessimistic, and often fail to accurately represent observed behavior. For example, a common class of concurrent algorithms, known as lock-free algorithms, provides system-wide progress, but cannot guarantee that any specific process will succeed in its operations. Interestingly, these are often the algorithms of choice in practice; despite their weak theoretical progress guarantees, they perform quite well in the majority of systems, displaying quick progress for all participating processes.

This is reminiscent of the gap between theory and practice in other areas of computer science. For example, many problems, like SAT solving and clustering, are known to be NP-hard, but are solved efficiently by practitioners every day. To explain this, a different approach in theoretical computer science, known as Beyond Worst Case Analysis [257], restricts the allowable inputs or input distributions. In concurrency, the difficulties stem from the worst-case adversarial scheduler, so a 'beyond worst case' approach in concurrency means restricting the power of the

scheduler.

For example, some work aims to relax the adversarial scheduling assumptions by replacing the adversary with a *random* scheduler [15, 16], which picks the next process to take a step according to some predetermined distribution over the processes. However, since real systems are not actually random, such models can misrepresent the schedules that are likely to arise in practice. Furthermore, even random schedulers cannot explain some basic practical phenomena that stump adversarial analysis.

1.1.1 Analyzing Backoff

An interesting example of practical algorithms whose success cannot be explained by existing models is in algorithms using exponential backoff. Backoff protocols, in which processes wait for increasing amounts of time before retrying a failed operation, are known to significantly improve the performance of many concurrent algorithms in practice [20, 148]. The idea behind backoff is to reduce contention; when many processes attempt to access the same memory at the same time, they are all delayed, and most of the attempted accesses fail. By waiting before retrying, a process is less likely to encounter contention again in its next attempt. However, while backoff is well understood in other settings [33, 52, 54, 117], shared-memory adversarial models cannot explain the practical advantages that backoff brings. An adversary could simply stall all processes until they all complete their backoff, and then proceed as if the backoff protocol were not there at all. Random schedulers have also so far failed to model the possibility that a process might prevent itself from participating in the protocol for a certain amount of time.

In Chapter 2, we address this issue by restricting the power of the scheduler in a different way. The first step in finding the right restrictions is to understand what causes asynchrony in the first place. The delays experienced by a process in a real system can be roughly divided into three categories; the *system* delays, caused by interrupts, the *hardware* delays, caused by cache coherence protocols and contention, and *self-imposed* delays, in which the algorithm itself waits, as is done in backoff protocols. Usually, all of these concerns are grouped into one, and modeled as the adversary; in our work, we separate them.

We define a *modular* model, in which at any point in time, each process belongs to exactly one of three sets, corresponding to the source of the delay it is experiencing. Processes move from one source of delay to another as they progress in the execution of their instructions, starting at self-delay before they begin executing a shared instruction, moving to system delay once a new instruction is invoked, then hardware delay as their instruction is being executed, and finally moving back to self-delay after they complete an instruction. Different policies govern the movement of processes among each of the categories of delay; if an adversary controls all three categories, this model boils down to the classic adversarial scheduler. However, the adversary can also be restricted to control just some of these categories, or can be given different power in each.

For the purpose of our analysis in Chapter 2, we instantiate the modular model with the following simple policies. Processes in the self-delay category control their own movement into the system-delay category; they move there whenever they invoke a new instruction. We let an adversary control the movement of processes from the system delay to the hardware delay. For hardware delays, we define *conflicts* between process instructions, and assume a FIFO priority

order; in every time step, every process p that does not conflict with anyone that started its hardware delay before p gets to complete its instruction and move back to the self-delay category. We say two instructions conflict if and only if they access the same location and one of them intends to modify that memory location.[1]

In real machines, the hardware delay stage blocks the process; it cannot do anything else while it experiences hardware delays due to contention. Thus, we charge the time a process spends in the hardware delay phase as *work* that the process was forced to do. The total work of an algorithm is calculated as the sum over all time steps of the number of processes in the hardware delay phase. We describe this model in more detail in Section 2.2.

We show that with this restriction on the allowable schedules, we can now analyze what has evaded rigorous characterization in the past; we provide the first analysis of backoff protocols in shared memory. In particular, we consider a simple contended scenario; n processes attempt to update the same location, each repeatedly trying until the first time it succeeds. We show that this simple protocol costs $\Omega(n^3)$ total work until all processes terminate if no backoff is used. However, if classic exponential backoff is applied, then $\Theta(n^2 \log n)$ work suffices. Thus, we are able to capture a theoretical separation between the naïve protocol and one that uses backoff.

Equipped with this insight, we then present a new backoff protocol that achieves better performance under our model than classic exponential backoff. The idea of this protocol is to make use of read instructions, which cause less conflicts and therefore spend less time in the hardware delay phase, to gauge the amount of contention in the system. In every new iteration of the protocol, a process flips a coin to determine whether to try to update the memory location or to simply read it. Its probability for heads is then adjusted depending on what it observed when accessing that location. We show that this simple idea improves on the work bound for this protocol; our algorithm, called *adaptive probabilities*, allows all processes to successfully update the same memory location within $O(n^2)$ work.

1.1.2 Measuring Real Schedules

The hardware delays that lead to contention are poorly understood, and can actually differ depending on the architecture on which an algorithm is run. While the hardware-delay policy we used in Chapter 2 provides one general way to model these delays, we could benefit from having a deeper understanding. Such an understanding could directly lead to better analyses, by abstracting it into a contention model and plugging it into the modular framework of Chapter 2.

To address this need, in Chapter 3 we present a tool, called *Severus*, that helps to view and analyze schedules produced by real hardware executions [50]. A user can run this tool on their own machine, and use the results to tailor their model assumptions to that machine to arrive at better predictive models. Through the development of this tool, and by analyzing the schedules of different machines, we also uncovered phenomena that are counter-intuitive, and were not documented before. In particular, we study contended executions on non-uniform memory access (NUMA) architectures. In NUMA architectures, which are now commonplace, cores are

[1]This conflict definition reflects cache coherence protocols, which allow a memory location to be held in 'shared mode' by several processes at once if they are all just reading the memory, but requires 'exclusive mode' for any process whose instruction will modify that location. Similar conflict models have been used in the past [116].

organized into groups called *nodes*, and each node has cache as well as main memory. All cores can access all shared caches and memory, through an interconnect network between the nodes. However, accesses to cache and memory in a core's own node (*local accesses*) are faster than accesses to the cache or memory of a different node (*remote accesses*).

We run Severus on two different NUMA machines; the first is an AMD machine with 8 NUMA nodes, and the second is an Intel machine with 4 NUMA nodes. Each machine employs a different cache coherence protocol. Interestingly, we show that, while the machines exhibit seemingly very different scheduling patterns, some key phenomena remain the same. For example, we show that in highly-contended workloads, a process's throughput *increases* when it accesses memory that is physically *far* from it (i.e., on a different NUMA node, thus requiring a *higher* latency). We also show that in general, regardless of how long an execution is run, the schedules on both machines are not *fair*, i.e., some processes are perpetually biased against, and get less opportunities to modify a contended memory location.

Creating a tool such as Severus is highly non-trivial. The difficulty can be thought of as a Schrodinger's Cat problem; the mere act of trying to observe the schedule of concurrent operations can significantly perturb it. Indeed, accessing a global timestamp to indicate the order of instructions can take tens of nanoseconds (whereas faster instructions take just a handful of nanoseconds) and can be highly contended. Thus, the order of instructions as documented by any timestamping mechanism is not very reflective of the order that would have occurred without the timestamps. We carefully avoid this problem by using the values that processes write into the contended memory itself to establish order; each process writes its own id, and uses a local log to document the id that was in the memory each time it reads that location. We later reconstruct a global ordering by merging the local logs. We believe that this work can lead to an understanding of how contention truly affects workloads, and can allow us to abstract our observations into an accurate contention model.

1.1.3 Leveraging Structure in Nested Parallel Programs

In Chapter 4, we take a different approach for improving the analysis of concurrent algorithms. We note that sometimes, understanding the underlying hardware and modifying the model accordingly isn't necessary for achieving meaningful analyses of concurrent algorithms; there may be other factors that limit the possible schedules that an algorithm must handle. Although concurrent shared memory can be chaotic and difficult to model, sometimes we use concurrent algorithms in a context that is more structured.

As a concrete example, consider concurrent algorithms that are used within a nested parallel program. Nested parallelism is widely used in many systems to take advantage of inherent parallelism in a task, and is broadly available in a number of modern programming languages and language extensions [79, 120, 124, 159, 161, 181, 199, 203]. In a nested parallel program, new parallel tasks can be created and removed using programming primitives, which govern the number of processes that may participate in the execution; `fork` and `async` create new parallel tasks, allowing new processes to join the computation to execute these tasks. Similarly, the `join` and `finish` primitives represent synchronization points, where parallel tasks are merged, terminating the processes that executed them. In this way, these programs impose a structure of their own on a concurrent execution.

Nested parallel computations can be modeled by a *series-parallel directed acyclic graph*, or *sp-dag*, where the vertices represent parallel tasks, and edges represent dependencies between them. A task cannot be executed unless all tasks that it depends on have completed; that is, all vertices with edges into a vertex v must complete their execution before v can begin. To efficiently run a nested parallel computation, an efficient sp-dag data structure must be implemented, which allows creating new tasks and keeping track of the number of dependencies remaining on each task.

In Chapter 4, we design an sp-dag structure that, when used in nested parallel programs, guarantees that all operations on it complete in amortized constant time, even when accounting for contention. Such a result was not known before; indeed, there was reason to believe that this would be impossible. The key algorithm behind an sp-dag data structure is a concurrent *counter* for the remaining dependencies on each vertex. This counter should be incremented by different processes when new dependencies are spawned, and decremented when dependent tasks are completed. However, there is a known lower bound showing that, when accounting for contention, any concurrent counter among n processes requires $\Omega(n)$ *stalls* for some process p. That is, in any execution, for some process p, $\Omega(n)$ processes must access the same location as p since p executed its previous instruction [116].

Crucially, for sp-dags, we do not need the full interface of a counter. In particular, the important part is only to know when a vertex in the dag has zero dependencies remaining, but we do not care about the precise count at all times. This allows us to circumvent the lower bound and use a data structure known as a scalable non-zero indicator (SNZI), which only gives an indication of whether or not the current count on it is zero [111]. While the SNZI data structure was previously introduced as a way to reduce contention when implementing counters, no contention analysis was provided. In Chapter 4, we extend and modify the original SNZI data structure and show that, when used in an sp-dag for nested parallel computations, our version of it guarantees a constant amortized number of contention stalls per increment and decrement. This data structure is the key component that makes our larger sp-dag data structure achieve these guarantees.

The work presented in Chapter 4 highlights the importance of remembering the context in which an algorithm will be run when designing and analyzing it.

1.2 Part 2: Non Volatile Memory

The theory of multiprocessor systems not only needs to accurately reflect technology that has existed for many years, but also must adapt to changing features. In Part II of this thesis, we consider non-volatile random access memory (NVRAM) and use theoretical models to find constructions that make use of its new features to improve concurrent algorithms.

NVRAM refers to a class of new main memory technologies that offer byte-addressable persistent memory at speeds comparable with DRAM. That is, upon a system fault (e.g., loss of power) in a machine with NVRAM, data stored in main memory will not be lost. This in contrast to previous non-volatile disks, which are around 3 orders of magnitude slower to access, and are accessed at a much coarser granularity. NVRAM now co-exists with DRAM on the newest Intel machines, and may largely replace or augment DRAM in the future. Thus, NVRAM yields a new model and opportunity for programs running on such machines—we could potentially

persist programs and data significantly faster than before.

However, without further technological advancements, caches and registers are expected to remain volatile, losing their contents upon a crash. This creates a challenge when programming with NVRAM for persistence. Simply running a program that was built for DRAM on NVRAM does not make it able to recover after a system fault, since some of its data would have been stored on registers and cache, and would therefore be lost. Furthermore, running programs built for persistence on disk doesn't work either; many updates can be done atomically on disk because of its the block-granularity, and updates never unintentionally overwrite its previous contents. This is not the case on NVRAM. Thus, the key question remains—how can we take advantage of persistent main memory to make a program recover after a fault?

There has been a surge of recent work designing algorithms and transactional memories that use NVRAM to recover after a system fault [74, 89, 92, 100, 122, 219, 278]. Redesigning programs to be persistent involves two parts; first, we must ensure that the state of memory after a fault is *consistent*, that is, could have resulted from an execution without a fault. However, even if the state of main memory is consistent, this might not be enough to be able to continue the execution after the system faults. In particular, when a program runs, it relies on key data like the program counter and the return value of function calls to determine what to do next. If this information is lost, the program cannot continue. Therefore, the second component for making programs persistent is ensuring that programs can continue their execution approximately where they left off.

In general, keeping the memory in a consistent state can be easy; for lock-free algorithms, for example, it is enough to flush every memory location accessed, immediately after accessing it [166]. However, we should aim to minimize injected flush instructions, since flushes are notoriously expensive. Nevertheless, we must be careful not to flush too little, since this can lead to an inconsistent state of memory, from which it is impossible to recover after a crash. The problem of flushing just enough to maintain correctness has been the subject of significant work in recent years [89, 100, 122, 123, 202].

1.2.1 Delay-Free Persistent Simulations

In Chapter 5, we focus on addressing the second issue; how to ensure that we can continue the execution of a persistent program after a system fault, even if we can ensure that the state of memory is consistent. To decouple the two concerns, we adopt the Parallel Persistent Memory (PPM) model of Blelloch et al. [59]. This model assumes that all shared memory accesses are done directly on persistent memory, thereby guaranteeing that the memory is in a consistent state at all times. However, the model assumes that each process owns a private *ephemeral* memory that, like caches and registers on real machines, loses its contents upon a fault. The process can write any local variables on its ephemeral memory, and register values like the program counter and return values of shared-memory instructions are stored on its ephemeral memory. The process can decide to persist values in its ephemeral memory by writing them on persistent memory.

The problem of continuing the execution after a fault can be challenging, especially in the concurrent setting. Concurrent algorithms are designed under the assumption that each instruction in them is executed exactly once. Repeating an instruction or skipping it altogether can

easily damage the safety of an algorithm. Furthermore, when designing persistent algorithms that can continue executing after a fault, it is desirable to minimize the changes that are made; decades of research into concurrent algorithms have led to designs that are highly optimized. It is therefore preferable to keep the old algorithms as a base, and change their structure as little as possible.

Our approach to solving this problem is to create *persistent linked simulations*, which replace each instruction of the original non-persistent version of an algorithm with a *simulation* that has the same effect. This helps maintain the original algorithm's structure. The simulation ensures that instructions are only repeated after a fault if it is safe to do so. Our main result is showing that for any concurrent program written with compare-and-swap (CAS) and read instructions, there is a simulation of it that introduces only constant overhead, and allows recovering to the same place in the execution within a constant number of steps. That is, our simulations are virtually *delay free*. We also show how to optimize this simulation to be efficient in practice, by reducing the overhead as much as possible.

1.3 Part 3: Remote Direct Memory Access

Another interesting technology that offers non-traditional features is Remote Direct Memory Access (RDMA). Although RDMA has been available for a while, it has mostly been used just for high performance computing, and has only recently been adopted in data centers. RDMA allows fast read and write access to a remote machine's memory by bypassing the remote CPU. Given this capability, researchers are working on understanding how to best use RDMA for fast communication. Many recent papers in the systems community have shown large performance benefits gained by replacing standard messaging technology with RDMA-based versions [42, 106, 171, 173, 250, 279]. However, it has so far been unclear whether these speedups are due to a fundamental difference between RDMA and previous hardware generations. In fact, recent work has shown that classic messaging remote procedure calls can be adapted to achieve almost the same performance as new RDMA-based algorithms [174].

In Part III of the thesis, we make steps towards a theoretical understanding of RDMA technology. Notably, this part of the thesis departs from the concurrent shared-memory setting in favor of a distributed setting in which processes are on physically separate machines. Such a setting is usually modeled with *message passing*, where processes communicate by sending and receiving messages over an asynchronous network. However, RDMA challenges this assumption, since it allows processes to also communicate via direct memory accesses. We therefore first focus on faithfully modeling RDMA's capabilities.

When designing a new theoretical model, it can be difficult to decide what details to account for, and which ones should be ignored in the name of keeping the model clean, intuitive, and useful. In this thesis, we focus on modeling the shared-memory power of remote memory accesses, as well as some other key features: RDMA's ability to dynamically open and destroy connections, its secure nature, and the cost of maintaining multiple connections. We assume we have a fully-connected message-passing network, on top of which we are allowed to pick shared-memory edges (representing RDMA links) to add, thus making it a *hybrid* network. We also allow algorithms to specify different access permissions to different parts of the memory, and

to change the shared-memory edges dynamically. We call this model the *message-and-memory*, or *M&M*, model, as it unites two very well studied communication models from the literature; message passing and shared memory. Interestingly, the simplicity of the M&M model means that it can be applicable not only to RDMA, but to several other memory and message technologies, including disaggregated memory and GenZ.

Using the M&M model, we show that there is a *fundamental* advantage to using an RDMA network over message-passing-only networks. We do so by considering the classic *consensus* problem as a lens through which to study RDMA. In consensus, the goal is for several processors to agree on the same output, despite network asynchrony and the possible failures of some of the processes. Well known bounds have shown that in message passing, consensus cannot be solved if half or more of the processes in the network have crashed, or if a third or more of the processors in the network experience *Byzantine* failures (i.e. behave maliciously). In Chapters 7 and 8, we present algorithms in the M&M model that solve consensus in a more fault-tolerant manner; we can tolerate more than half of the network processes crashing, and more than a third of the network processes being Byzantine. The focus of these two chapters is slightly different, with Chapter 7 considering how to use RDMA while scaling to large networks, and Chapter 8 focusing on harnessing the full power RDMA can bring to a small well-connected network.

1.3.1 Scaling to Large Networks

In Chapter 7, we consider how to increase the fault tolerance of consensus algorithms in large message-and-memory networks.

Shared-memory systems have worse scalability than message-passing systems due to hardware limitations. For example, a typical shared-memory system today has tens to thousands of processes, while message-passing systems can be much larger (e.g., map-reduce systems with tens of thousands of hosts [102], peer-to-peer systems with hundreds of thousands of hosts, the SMTP email system and DNS with hundreds of millions of hosts, etc). This limitation is also apparent in RDMA networks. If a single node in the network maintains many open RDMA connections, significant slowdowns can be observed for RDMA operations on that node.

To model efficient networks with many nodes, we therefore think of an RDMA-enabled system as a system in which all machines can communicate with each other over message passing (all-to-all links), but only subsets of the system can share memory. Under this model, we consider the problem of crash-tolerant consensus, and show that even with a few shared-memory connections, consensus can be solved with higher fault tolerance than is achievable with message passing alone.

Our approach is to first show a consensus algorithm that operates on an M&M network, making use of shared-memory edges that are available, but which guarantees correctness regardless of how many shared-memory connections it has. The key insight is that shared memory can solve crash-tolerant consensus in a system where arbitrarily many processes may fail, whereas message-passing networks require a majority of the processes to be active. We therefore run a message-passing algorithm in the network, but have each process run a shared-memory consensus algorithm with its shared-memory neighbors in each round to agree on their values. Each process then sends not only its own message, but also the messages of its neighbors.

Thus, the fault tolerance of our algorithm depends on the shared-memory connections of

the graph. In particular, we show that the fault tolerance achieved in a given M&M network G depends on the *vertex expansion ratio* of G. The vertex expansion ratio of a graph is a measure of how well connected the graph is. There are known graph constructions that guarantee high vertex expansion and low maximum degree. Thus, using such a construction to choose the shared-memory connections in an M&M network yields a highly fault tolerant, highly scalable consensus algorithm.

1.3.2 Dynamic Permissions in Small Networks

In Chapter 8, we focus on small networks that can handle all-to-all RDMA connections without slowdowns. We consider the power that RDMA can bring not only from simply sharing memory, but also from its ability to provide dynamic access permissions to different parts of the memory. We show that RDMA can improve the fundamental trade-off between fault tolerance and performance that exists in message-passing and shared-memory algorithms.

In particular, each process in an RDMA network can divide its memory into *regions*, and can choose, for each region r and each process p in the system, whether to give *read* access, *write* access, *both*, or *neither* for p to access r. We study how this feature helps in solving two variants of consensus; a crash-tolerant version like the one considered in Chapter 7, and a Byzantine-tolerant version, which tolerates processes that become malicious and actively try to break the protocol. For both types of consensus, we consider two metrics; the number of process failures that can be tolerated, and the *best-case*, or *common-case*, running time that is achievable. Common-case running time is a measure of how quickly a process can reach a decision in an instance of consensus in an execution in which no failures happen and messages are synchronous. However, the protocol must be able to tolerate failures and asynchrony, and can't know a priori that it will be experiencing a good run. This way of measuring the performance of a consensus algorithm is important in practice, since, while protocols must be designed to tolerate worst-case conditions, most of the time the network is well-behaved.

We show that dynamic permissions in RDMA can be leveraged to achieve better common-case performance than what is possible in shared memory systems, while achieving better fault tolerance than what is possible in message-passing systems. This is true for both the crash-tolerant and the Byzantine-tolerant versions of the problem. That is, RDMA can leverage its shared memory and permission features to get the best of both worlds between shared memory and message passing.

More specifically, we measure common-case running time by the number of *network delays* it takes until the *first process* reaches a decision on the consensus value. A network delay is the amount of time it takes for a message sent by some process p to be received by some process q. Since the network might be asynchronous, the actual time a single network delay takes may vary greatly. However, network delays give a good way to measure the amount of communication required in a protocol, without relying on specific network parameters.

For Byzantine-tolerant consensus, we show that it is possible to have an algorithm that tolerates f Byzantine failures in a system with $n \geq 2f + 1$ processes, that reaches it decision withing *two network delays*. The best possible solution in message passing can only work in a network with $n \geq 3f + 1$ processes, and can reach its decision also within two network delays [31]. In shared memory, it was not known how to solve Byzantine consensus. We show it is possible to

solve with $n \geq 2f + 1$, but would require at least four network delays. Similarly, for crash-tolerant consensus, we present an algorithm in the M&M model that tolerates $f < n$ failures and terminates in two network delays in the common case. This matches the best possible fault tolerance of shared memory and the best possible performance of message passing, while each of the classic model cannot match our algorithm in the other metric.

For both crash- and Byzantine-tolerant consensus, the key to achieving this high common-case performance is the ability to *dynamically change permissions*. In both, this ability is used to give *exclusive write access* to some memory region to a single process at a time. That process is considered the *leader* while it holds the exclusive write permission. If the leader is suspected by some other process to have failed, its exclusive permission is revoked. However, if not, the leader can reach a decision very quickly, as long as it succeeds in writing its value without its permission being revoked.

Thus, in Chapters 7 and 8, we prove that RDMA is indeed more powerful than classic communication primitives, thereby giving strong motivation to continue to use and develop RDMA technology. Follow-up work uses insights from the results presented in Chapter 8 to design a state machine replication system that significantly outperforms competitors [14].

1.3.3 State Machine Replication with RDMA

In Chapter 9, we put our theory to the test, and implemented an RDMA-based state machine replication (SMR) system using the ideas developed in our theoretical work. In particular, in Chapter 8, we showed that crash-tolerant consensus can be solved in RDMA in two network delays, which can also be thought of as *a single round trip*, in the common case. We use this insight to guide our design of the SMR system.

To understand SMR, consider the following client-server setting: a server maintains a state machine, and handles client requests in the form of state machine operations. The server orders client requests and executes them on the state machine, replying to the client with the output of their state machine operations. In this setting, if the server experiences a failure or slowdown, the entire system becomes unavailable, with client requests going unanswered, until the server is recovered. In many applications, the risk of long periods of unavailability is too high. In such cases, *state machine replication (SMR)* is often implemented to boost availability. The idea behind SMR is to have not one, but several servers, which replicate client requests among themselves and coordinate to agree on the order in which these requests should be serviced. Then, each server locally keeps an up-to-date copy of the state machine. Now, if one server fails, the other servers can take over the execution, masking most of the recovery time from the client.

At its core, an SMR system relies on a consensus protocol that is executed among the servers to agree on requests. This is where insights from our theoretical work comes in, since we studied consensus in RDMA and understood how to design a highly efficient algorithm in Chapter 8.

However, unlike the single-shot consensus protocol presented in Chapter 8, an SMR consensus protocol is long-lived. That is, instead of agreeing on a single value, the consensus protocol is continuously used to agree on consecutive values as more requests enter the system. While a single-shot consensus protocol can be repeatedly used to implement a long-lived version, such an approach neglects many opportunities to optimize along the way. Furthermore, while the consensus algorithm of Chapter 8 only ensured that the *first* process reach a decision quickly, in an

SMR system, it is important that all processes know the decision value and update their state machines accordingly. The protocol used for Mu, the SMR system in Chapter 9, therefore uses insights from the theoretical work, but is carefully designed to optimize for practical concerns.

Indeed, we optimize Mu not only for common-case executions, in which we ensure the latency is just that of a single RDMA write operation, but also for *fail-over time*. That is, we ensure that when a server, and, in particular, the *leader server* of the protocol fails, a new leader takes over as quickly as possible. Fail-over time is composed of two factors; the first is *failure detection*, i.e., recognizing that a failure happened in the first place, and the second is *leader election*, i.e., choosing a server to be the new leader and having that new leader restart the execution. Failure detection depends on the *variance* in network and process latency that is expected, since detecting failures too eagerly can result in false positives. We develop a failure detection mechanism that depends only on process speed variance and not that of the network, thereby allowing us to detect failures much more quickly than other systems. For leader election, our main bottleneck is switching RDMA permissions, as discussed in Chapter 8. We study various ways of implementing permission switches in RDMA, and show it is possible to accomplish in only a couple of hundreds of microseconds.

Our resulting implementation outperforms state-of-the-art RDMA-based competitors by a factor of $2.5-6\times$ in the common case, and by more than an order of magnitude when recovering from server failures. This highlights the impact that reasoning about algorithms in theory can have on practical performance.

1.4 Summary of Contributions

In summary, this thesis narrows the gap between theory and practice in concurrent and distributed computing. Its key contributions are as follows.

Analyzing Concurrent Algorithms. We improve our ability to accurately analyze the running time of concurrent shared-memory algorithms. This is achieved through three approaches: (1) we present a new modular model for calculating contention costs in shared-memory algorithms and demonstrate its effectiveness by analyzing and improving exponential backoff; (2) we present a tool for tracing the operation schedule produced on real machines in a variety of workloads; and (3) we demonstrate that accurate analysis of concurrent algorithms is possible in structured settings by designing and analyzing a provably efficient series-parallel dag structure for nested parallel computations.

NVRAM. We study the Parallel Persistent Memory (PPM) model of NVRAM [59], and present the first general simulation that can convert any concurrent algorithm that uses atomic Read and CAS into a persistent algorithm. The PPM model decouples concerns of memory consistency after a fault from *detectability* [122], or the ability to continue an execution after the fault. Thus, our simulation is a general way to achieve detectability for concurrent algorithms. We introduce the notions of computation and recovery delays as measurements of the overhead of such a simulation, prove that our simulation is both computation- and recovery-delay free, and present optimizations that make the simulation more practical and work for a large class of algorithms.

	Message Passing	Shared Memory	**RDMA (large networks)**	**RDMA (small networks)**
Crash fault tolerance	$f < n/2$	$f < n$	$\mathbf{f < (1 - \frac{1}{2h}) \cdot n}$	$\mathbf{f < n}$
Byzantine fault tolerance	$f < n/3$	$\mathbf{f < n/2}$	?	$\mathbf{f < n/2}$
Complexity	2	$\mathbf{\geq 3}$	?	**2**

Table 1.1: Summary of previously-known and new RDMA results on consensus fault tolerance and complexity. Comparing the message-passing and shared-memory models to the M&M RDMA model. New results presented in this thesis are in blue. Complexity is measured in message delays in common-case executions. n and f represent the number of processes in the system and the maximum number of failures, respectively, and h is the vertex expansion ratio of the underlying shared memory graph in an RDMA network.

RDMA. Finally, we present the *Message-and-Memory (M&M)* model, a model that combines features of the message-passing and the shared-memory models, and which captures the capabilities of RDMA in data centers. We prove that the M&M model can tolerate more failures when solving consensus than is possible in the message-passing model, and that solutions for the consensus problem can be more efficient and more scalable in the M&M model than in shared memory. Along the way, we prove new results for the Byzantine fault tolerance and efficiency of consensus in the shared-memory model. Table 1.1 summarizes these results. Furthermore, we demonstrate that the M&M model indeed captures practical concerns by implementing an RDMA-based state-machine replication (SMR) system using the insights gained from our theoretical study of the M&M model. Our SMR system, Mu, can replicate requests at least $3\times$ faster than other state-of-the-art systems, and can recover from server failures at least an order of magnitude faster.

1.5 Model and Preliminaries

This thesis considers the shared memory concurrent setting and some of its variants. In this section, we present the classic shared memory model, which will form the basis for all the models we consider.

Processes and Instructions. We assume a system of n asynchronous processes, $P = \{p_1 \dots p_n\}$, that communicate with each other by executing machine instructions on shared memory. Shared memory machine instructions may be atomic read, write, compare-and-swap (CAS), test-and-set (TAS), or fetch-and-add (FAA) instructions. A $write(X, val)$ updates the contents of memory location X to val. A $CAS(X, old, new)$ updates the contents of memory location X to new if it contains old, and returns true in this case. Otherwise, the contents of X do not change, and the CAS returns false. We say that a CAS is *successful* if it returned true. A $FAA(X, num)$ increments the value in X by num, and returns the value stored in X before the increment. A $TAS(X)$ sets the value of X to 1, and returns the previous value of X. The TAS instruction

operates on booleans; the value of X may only ever be 0 or 1. An atomic $read(X)$ returns the value most recently written by a write, TAS, FAA, or successful CAS instruction on memory location X. Other shared memory instructions are possible, but are not considered in this thesis. Processes may also execute *local* read and write instructions. Local instructions are non-racy; if a process p_i executes a local instruction on memory location ℓ, ℓ cannot be accessed by any other process until p_i explicitly *makes ℓ shared*. The execution of an instruction by a process p_i is sometimes referred to as p_i taking a *step*.

The ABA-problem. Usually, CAS instructions are used in conjuction with read instructions; a memory location X is read by process p, and the return value of the read instruction is then used as the 'old' value argument for the following CAS instruction by p on X. It is often convenient to think of a CAS instruction as failing if and only if X's value did not change since p's most recent read of X. However, this does not always hold; it could be the case that after p read a value v from X, X's value changed to some v' and then back to v before p executed its CAS. This scenario is called the *ABA-problem*[2]. To be more precise, we say that the ABA-problem occurs if a value that already appeared in some memory location X is written or CASed into X again at a later time. We note, however, that this scenario can be avoided in practice if a sequence number is attached to each value that is written or CASed on each location X, therefore preventing repeated values. Therefore, it is common to assume *ABA-freedom*, i.e., that the ABA problem cannot occur.

Executions. An *execution* is a sequence of atomic steps during the run of a program. There are a few levels of abstraction at which one can view an *execution*. We begin by discussing *low-level executions*. An low-level execution is a sequence of processes, representing the order in which processes took steps. One can also view this as a sequence of steps, where each step specifies the process that executed it, the memory location on which it was executed, the instruction that was executed, and its arguments. A *legal* low-level execution E is one in which each instruction i in E returns the correct value assuming that the state of the memory it accesses is the state that results from executing the sequence of steps in E up to i.

Objects. We often consider algorithms that implement concurrent *objects*, which can intuitively be thought of as data structures. An object has a *sequential specification*, which defines a set of *operation types* that can be called on it, and specifies the expected behavior of each operation type when executed sequentially. An *implementation* of an object is a set of algorithms, one per operation type of the object.

Processes interact with objects through *events*. There are two types of events: an *invocation* of an operation type op of an object o with input value val, denoted $inv(o, p_i, op, val)$, and the *response* of the last operation type invoked by p_i on o, with return value res, denoted $res(o, p_i, op, res)$.

High Level Executions and Operations. A *high-level execution* is a sequence of events. A response r is said to *match* an invocation i in a high-level execution E if r is the first reponse

[2]The name A-B-A represents the different values written in memory location X.

after i in E that has the same process, operation type and object as i. An *operation* in a high-level execution E is an invocation of an operation type in E with its matching response, if it exists. An operation can be thought of as an instantiation of an operation type[3]. We say an operation is *pending* in a high-level execution E if it has been invoked but has not received a matching response in E. An operation is *complete* in E if it has both an invocation and a matching response. Given two operations op_1 and op_2 in E, we say that op_1 *happens before* op_2, denoted $op_1 <_H op_2$, if op_1's matching response is before op_2's invocation in E. If neither $op_1 <_H op_2$ nor $op_2 <_H op_1$, we say that op_1 and op_2 are *concurrent*.

Note that an low-level execution that contains object operations can be mapped to a high-level execution by placing invocation and response events immediately before (resp. after) the first (resp. last) instructions of the corresponding operation, and then removing all steps. A single high-level execution could map to several low-level executions, where the steps of individual operations are interleaved differently. When the context is clear or unimportant, we refer to low-level executions and high-level execution collectively as executions.

Execution Projections. The *projection* of an execution E onto a process p_i, denoted by $E|p_i$, is the subexecution of E that contains exactly all of the steps or events that involve p_i. Similarly, the projection of E onto an object o, denoted by $E|o$ is the subexecution of E that contains exactly all of the steps or events that were applied on o (for steps in an low-level execution, o is a memory location rather than an implemented object). We say that a sequential high-level execution is *legal* if, for each object o, $E|o$ satisfies the sequential specification of o. Note that this definition of legality applies to high-level executions only, and is different from the definition of a legal *low-level execution*.

Properties of (High-Level) Executions. A high-level execution is said two be *sequential* if each invocation is immediately followed by a matching response. Note that in this case, the happens before relationship is a total order on the operations of a high level execution. We say that a sequential high-level execution is *legal* if, for each object o, $E|o$ satisfies the sequential specification of o. Note that this definition of legality applies to high-level executions only, and is different from the definition of a legal *low-level execution*. A *completion* of a high-level execution E is a high-level execution E' in which, for each pending operation in E, a matching response is appended at the end of E. We say that two low-level executions or high-level executions E and E' are *equivalent* if, for each process p_i, $E|p_i = E'|p_i$.

Linearizability. We say that a high-level executionis *linearizable* if there is a way to order the operations that respects their happens-before relationship and yields a legal sequential execution. More formally, execution E is *linearizable* if there exists a completion E' of E and a legal sequential execution S such that (1) S and E' are equivalent, and (2) $<_{E'} \subseteq <_S$ [155]. Intuitively, a linearizable high-level execution is one in which each operation *appears to take effect instantaneously at some point between its invocation and response.* An implementation of an object is

[3]In a slight abuse of terminology, we sometimes refer to operation types simply as operations when the context is clear.

linearizable if all possible executions that can be produced by processes executing its operation algorithms are linearizable.

Adversarial Scheduler. Asynchrony is modeled by an *adversarial scheduler*, which, at every point in the execution, determines which process will execute the next instruction. Formally, the adversary is a function from the set of low-level executions to the set of processes, which, given an low-level execution, outputs a process. The power of the adversary can be controlled by specifying the amount of information it gets as the input low-level execution; instead of specifying the instruction, location, process, and arguments of every step in the low-level execution, some subset of these can be specified, thereby giving the adversary less power. An *oblivious* adversary is one that recieves only the process id for each step in the execution. An *adaptive* adversary receives full information.

Progress Conditions. An implementation of an object o is *lock-free* if, in every infinite execution of the operations of o, infinitely many operations complete. An implementation of an object o is *wait-free* [149] if, in every infinite execution of the operations of o, every process completes infinitely many operations. Note that the key difference is that lock-freedom guarantees global progress in the system, since not all processes can get stuck forever, whereas wait-freedom guarantees local progress, in that no process can get stuck [150].

1.6 Bibliographic Note

Chapter 2 is based on

> Naama Ben-David and Guy E Blelloch. Analyzing contention and backoff in asynchronous shared memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–62. ACM, 2017

Chapter 3 is based on

> Naama Ben-David, Ziv Scully, and Guy E Blelloch. Unfair scheduling patterns in NUMA architectures. In *International Conference on Parallel Architecture and Compilation (PACT)*, pages 205–218. IEEE, 2019

Chapter 4 is based on

> Umut A Acar, Naama Ben-David, and Mike Rainey. Contention in structured concurrency: Provably efficient dynamic non-zero indicators for nested parallelism. *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017

Chapter 5 is based on

> Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2019

Chapter 7 is based on

> Marcos K Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–60. ACM, 2018

Chapter 8 is based on

> Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of RDMA on agreement. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 409–418, 2019

Chapter 9 is based on

> Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, Athansios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2020

In an effort to keep the thesis short, several other papers by the author on related topics have been omitted [44, 45, 46, 48, 123].

Part I

Contention in Shared Memory

Chapter 2

Analyzing Backoff

2.1 Introduction

Exponential backoff protocols are commonly used in many settings to avoid contention on a shared resource. They appear in transactional memory [261], communication channels [52, 53, 117, 118], radio and wireless networks [33, 35], local area networks [230], and even shared memory algorithms [20, 148]. Many backoff protocols have been developed and proved efficient for these communication tasks. The general setting is that many processes want to send a message over a channel (or a network), but the transmission fails if more than one message is sent at the same time. Usually, transmissions are assumed to happen in synchronous rounds, and time is measured by the number of rounds until all messages are successfully transmitted. A round is wasted if more than one process attempts to transmit, but the cost remains constant regardless of how many processes attempted to transmit in that round.

The situation is different in a shared memory setting. If many processes attempt to access the same memory location, they all suffer *contention*, which slows down their response time, and becomes worse as the number of conflicting processes increases. For example, if n processes all access the same memory location at the same time, they queue up behind one another, causing $\Theta(n)$ 'wait time' for each process on average, for a total of $\Theta(n^2)$ total wait time, or work. Several theoretical models have been developed to try to capture these costs [110, 113, 131, 132, 146]. However, none of these models have been applied to analyze backoff protocols. The reason is, at least in part, that these models are not well suited for the task. In general, shared memory contention and performance are more difficult to model than channel communication because of the inherent asynchrony in shared memory systems.

Asynchrony in shared memory algorithms is usually modeled by assuming a powerful adversary that determines the schedule of instructions. Known models capturing contention, like those of Dwork, Herlihy, and Waarts [110] and Fich, Hendler and Shavit [113], while accounting for contention cost, give too much power to the adversary to be able to get any meaningful worst case bounds on backoff protocols. The model of Dwork *et al.* assumes every instruction has an invocation and a response, and each instruction suffers contention equivalent to the number of responses to other instructions that occurred between its own invocation and response. Fich *et al.* assign cost differently; in particular, they assume only modifying instructions that immedi-

ately precede a given instruction i can have an effect on i's running time. However, both models still allow an adversary to control when instructions' responses appear in the execution. This means that while these models are reasonable for obtaining lower bounds, they are impractical for attaining meaningful upper bounds. Because of the great power the adversary is given, the advantage of backoff protocols is lost; the adversary can simply wait for processes that are backing off to invoke their next operation, at which point it regains full control, and can determine the conflicts in the execution with as much flexibility as if no backoff was ever attempted.

Further work by Hendler and Kutten [146] restricts the power of the adversary in an attempt to capture more likely executions. They present a model similar to that of Dwork *et al.* [110], but add the notion of k-synchrony on top of it. This requirement states that an execution is k-synchronous if in any subexecution, no process can do $k+1$ operations before all other processes do at least one. This property, while attractively simple, does not succeed in capturing all possible real executions. Asynchrony in real systems often arises from processes being swapped out. In such cases, one process can be delayed a great deal with respect to the others, and no reasonable value of k allows for such delays.

All of these models have an additional problem: there is no clear notion of time. This further complicates the analysis of time-based protocols such as exponential delay, where processes delay themselves for increasingly longer amounts of time.

A different line of work aiming to create a more realistic model for shared memory has focused on relaxing the adversarial scheduler to a stochastic one, leading to tractable analysis of the step complexity of some lock-free algorithms [15, 16]. However, this line of work also relies on strong assumptions on the behavior of the scheduler. In the next chapter, we study the schedules produced in practice and show that such assumptions do not always reflect real executions. Furthermore, such stochastic assumptions on the adversary have so far not directly accounted for contention, rendering the analysis of backoff under this model meaningless.

In spite of the difficulty of theoretically analyzing backoff, these protocols have been shown to be effective in combatting shared memory contention. Many systems benefit greatly from the application of an exponential backoff protocol for highly contented memory [20, 135, 148, 227, 233]. For about 30 years, exponential delay has been successfully used in practice to significantly reduce running time in algorithms that suffer from contention in shared memory.

In this chapter, we take a first step towards closing this gap between theory and practice. We define a model for analyzing contention in shared memory protocols, and use it to analyze backoff. Our model allows for asynchrony, but maintains a clear notion of time. We do this by giving the adversary full power on one part of the processes' delay, but taking it away in another part. In particular, we allow each process to have up to one *ready* instruction, and the adversary determines when to invoke, or *activate*, each ready instruction. However, once active, instructions have a priority order that the adversary can no longer control. This models the difference between delays caused by the system between instructions of a process (e.g. caused by an interrupt), and delays incurred after the instruction has been invoked in the hardware (assuming an instruction cannot be interrupted midstream). We then define *time steps*, in which all active instructions attempt to execute. Furthermore, we define *conflicts* between instructions; two conflicting instructions cannot be executed in the same time step. An active instruction will be executed in the first time step in which it does not conflict with any higher-priority instruction. In this chapter, we define conflicts as follows: modifying instructions (such as compare-and-swap) conflict with all

other instructions on the same location, but any two non-modifying instructions do not conflict. This reflects the fact that in modern systems, multiple reads to the same location can go through in parallel, but a modifying instruction requires exclusive access to the memory, sequentializing other instructions behind it. This is due to cache coherency protocols. which allow cache lines to be read in shared mode by several processes at once. However, our model allows any definition of conflicts to be plugged in, and can therefore adapt to different architectural concerns.

We then consider a situation in which n processes want to update the same memory location. Such a situation arises often in the implementations of various lock-free data structures such as counters and stacks. We use our model to analyze various protocols that achieve such an update. We first show that a naïve approach, which uses a simple read-modify-write loop with no backoff, can lead to a total of $\Omega(n^3)$ work, while a classic exponential delay protocol improves that to $\Theta(n^2 \log n)$. To the best of our knowledge, this is the first analysis of exponential delay for shared memory. Perhaps surprisingly, we show that this protocol is suboptimal. We propose a different protocol, based on adapting probabilities, that achieves $O(n^2)$ work.

The key to this improvement stems from using read operations to get a more accurate estimate of the contention that the process is facing. Recall that in our model, reads can go through in parallel, while modifying instructions conflict with others. In our protocol, instead of delaying for increasingly long periods of time when faced with recurring contention, a process becomes increasingly likely to choose to read rather than write. Intuitively, reading the value of a memory location allows a process to gauge the level of contention it's facing by comparing the new value to the previous value it read. Each process keeps a write-probability that it updates after every operation. It uses this probability to determine its next instruction; i.e. whether it will attempt a compare-and-swap or only read. If the value of the memory does not change between two consecutive reads, the process raises its write-probability. Otherwise, the process lowers it. This constant feedback, and the possibility of raising the probability back up when necessary, is what allows this protocol to do better than exponential delay.

In summary, the contributions presented in this chapter are as follows:

- We define a new time-based cost model to account for contention in shared memory;
- We provide the first analysis of the commonly used exponential delay protocol; and
- We propose a new backoff protocol that is asymptotically better than the classic one.

2.2 Model

Our model aims to separate delays caused by contention from other delays that may occur in shared memory systems. In most known shared memory models, all types of delays (or causes of asynchrony) are grouped into one, and are treated by assuming an adversary that controls the order of execution. However, we note two predictable sources of delay: (1) processes often engage in local work that does not affect the scheduling of others, and (2) once an instruction is invoked in hardware, while its performance can greatly vary due to the architecture and contention, it is relatively predictable. In this chapter, we model this delay in a simple way, and in the next chapter, we delve into how architectural design can affect it. Since local instructions do not affect the scheduling of other processes, in this chapter, we use the word *instructions* to refer

to only those on shared memory.

Each process $p \in P$ may have up to one pending instruction $I_t(p)$ at any time t. p may be in one of three states at any point in time, depending on its progress in executing its current instruction: *idle*, *ready*, or *active*. p is idle if it currently has no pending instruction. This could mean that p is executing some local work (recall that we don't consider local work as instructions), or that p is executing the delay prescribed by some backoff protocol. Once p finishes its local work, it invokes an instruction and moves to the ready state. The ready processes represent processes with instructions that are ready to run, but have not yet been invoked in the hardware; the instruction might be delayed because the calling process is swapped out, failed or otherwise delayed by the system. Finally, a process is active if its instruction has already started executing in the hardware, and is now delayed by contention (e.g. executing the cache coherency protocol).

In this way, the processes in the system at a time t are partitioned into three subsets: an unordered set of *idle* processes D_t, an unordered set of *ready* processes R_t and an ordered set of *active* processes A_t. Recall that each process p in R_t and A_t has exactly one instruction i_p, t associated with it. We therefore sometimes refer to the elements of R_t or A_t as instructions instead of processes. There is a priority order, denoted $<$, among instructions in A_t. Furthermore, to model contention, we define *conflicts* between instructions. For any pair of operations $i, j \in I_t$, the predicate $C(i, j)$ is *true* if i conflicts with j, and is *false* otherwise.

In each time step t, each process may undergo at most one of the following transitions: it may move from D_t to R_{t+1}, from R_t to A_{t+1}, or from A_t to D_{t+1}. The factors that determine each transition are different for each set. Moving from D to R is determined by the process itself; when a process completes its local work, it may invoke its next shared memory instruction, thereby moving itself from D to R. When a process is in the ready set, its movement is controlled by an adversarial scheduler. More specifically, the adversary chooses a (possibly empty) subset $S_t \in R_t$ to activate. It gives a priority order among the instructions in S_t, and adds them to A. There is a *merging policy* that determines the priority of the elements of S_t with respect to the instructions already in A_t. This merging policy is deterministic, and maintains the invariant that if, for some instructions $i, j \in A_t$, $i < j$ and both i and j are in A_{t+1}, then $i < j$ in A_{t+1}. That is, regardless of the new instructions added to A in time step t, the relative priorities of A_t do not change. Similarly, elements of S_t maintain their relative priorities assigned to them by the adversary when they are merged into A. Thus, the adversary can determine the priorities of instructions when it activates them, but has no control over them once it does so.

Finally, in time step t, we define the set of *executed* instructions to be $E_t = \{i \in A_t | \nexists j \in A_t . C(i, j) \wedge j < i\}$. That is, an instruction $i \in A_t$ is executed in time t if and only if there is no other instruction in A_t that conflicts with i and is ahead of i in the priority order. The processes that invoked the instructions in E_t are the ones that transition from A_t to D_{t+1} at time t.

For the purpose of this chapter, we say that $C(i, j)$ is true if and only if i and j both access the same memory location and at least one of them is a modifying instruction. This function can also represent other types of conflicts, such as false sharing, depending on the behavior of the system we want to model. Furthermore, the merging policy considered in this chapter is simple FIFO ordering; the instructions of S_t get added behind all instructions already in A_t.

Execution historiess are defined similarly in this model as they are in Section 1.5. An *execution history*, or simply *execution*, E is a sequence of the instructions executed by processes in the

system, which defines a total order on the instructions. This total order must be an extension of the partial order defined by time steps; that is, for any two instructions i and j, executed in time t and t' respectively, if $t < t'$ then i appears before j in E.

We consider two measurements of efficiency for algorithms evaluated by this model. One measurement is immediate: we say the running time of an execution is the number of time steps taken until termination. Another measurement is called *work*. Intuitively, this measures the total amount of cycles that all processes spent on the execution. We say that a process spends one cycle for every time step t in which $I_t(p) \in A_t$. Thus, the amount of work taken by an execution is given by the sum over all time steps t of the size of A_t (i.e. $W = \sum_t |A_t|$). All of the bounds in this chapter are given in terms of work. We feel that work better represents the demands of these protocols on the resources of the machine, as inactive processes could be asleep or making progress on some unrelated task.

Adversary. We assume an oblivious adversary [32]. That is, the adversary does not see processes' local values before making them active, and thus cannot make decisions based on that knowledge. Such values include the results of any random coin flips and the type of operation that the process will perform. The adversary can, however, know the algorithm that the processes are following and the history of the execution so far. We also assume that once an instruction is made active, the adversary may see its local values. However, at that point, the adversary is no longer allowed to make changes to the priorities, and thus cannot use that knowledge.

Protocols. In the rest of this chapter, we analyze and compare different backoff protocols. To keep the examples clean and the focus on the backoff protocols themselves rather than where they are used, we consider a simple use case: a read-modify-write update as defined in Algorithm 2.1.

An *update attempt* by process p is one iteration of Algorithm 2.1. In every update attempt, p reads the current value, then uses the *Modify* function provided to it to calculate the new value it will use for its CAS instruction. The Modify function can represent any change a process might need to make in an algorithm that uses such updates. A *successful update attempt* is one in which the CAS succeeds and the loop exits. Otherwise, we say that the update attempt *failed*. For simplicity, we assume that every CAS instruction has a distinct v_n, and thus an update attempt fails if and only if there was a successful CAS by a different process between its read and its CAS. This assumption is also known as ABA-freedom. Note that such an update protocol is commonly used in the implementation of shared objects such as counters and stacks.

```
1 update(x, modify){
2   while true { //one iteration is an update attempt
3     currentVal = Read(x);
4     val = modify(currentVal);
5     if CAS(x, currentVal, val){ return; } } }
```

Figure 2.1: Update Loop

```
1 update(x, modify){
2   maxDelay = 1;
3   while true { //one iteration is an update  attempt
4     currentVal = Read(x);
5     val = modify(currentVal);
6     if (CAS(x, currentVal, val)){ return;}
7     maxDelay = 2· maxDelay;
8     d = rand(1, maxDelay);
9     wait d steps; } }
```

Figure 2.2: Exponential Delay

2.3 Exponential Backoff

Exponential backoff protocols are widely used in practice to improve the performance of shared-memory concurrent algorithms [20, 148, 227]. However, to the best of our knowledge, there has been no running time analysis of these algorithms that shows why they do so well. In this section, we provide the first analysis of the standard exponential delay algorithm for concurrent algorithms.

In general, the exponential backoff protocol works as follows. Each process starts with a maxDelay of 1 (or some other constant), and attempts an update. If it fails, it doubles its maxDelay, and then picks a number d uniformly at random in the range $[1, \text{maxDelay}]$. It then waits d time units, and attempts another update. We refer to this approach as the *exponential delay* protocol.

Exponential delay is common in the literature and is applied in several settings to reduce running time; for example, Mellor-Crummey and Scott use it for barriers [227], Herlihy uses it between successive attempts of lock-free operations [148], and Anderson used it for spin locks [20]. The pseudo-code for exponential delay on an update protocol is given in Algorithm 2.2.

Using our model, as defined in section 5.2, we show that in an asynchronous but timed setting, the exponential delay protocol does indeed significantly improve the running time. In particular, we show that for n processes to each successfully update the same memory location once, the naïve (no-backoff) approach requires $\Theta(n^3)$ work, but using exponential delay, this is reduced to $\Theta(n^2 \log n)$. For simplicity, we assume that each process only needs to successfully write once; after its first successful write, it will not rejoin the ready set. To show the upper bound for exponential delay, we rely on one additional assumption, which is not used in the proofs of the lower bounds.

Assumption 2.3.1. *In any time step t where the active set A_t is empty and the ready set R_t is not, the adversary* must *move a non-empty subset of R_t to A_t.*

Analyzing the Naïve Protocol. We consider n processes trying to update the same memory location using a naïve update loop with no backoff. The pseudocode of this protocol is given in Algorithm 2.1. To give a lower bound for this protocol, we imagine a very simple adversarial behavior; whenever any instruction is ready, the adversary immediately activates it. Note that this is not a contrived or improbable adversary. Nevertheless, this scheduling strategy is enough

to show a $\Omega(n^3)$ work bound.

The intuition is as follows: the active set has around n instructions at all times. A successful update attempt can only happen about once every n time steps, because CAS attempts always conflict, and the active queue is long. The average process will thus have to make around $n/2$ update attempts before succeeding, and wait around $n/2$ time steps for each one, thus needing $\Omega(n^2)$ work. This gives us the bound of $\Omega(n^3)$ work for all processes together.

Theorem 2.3.2. *The naïve update protocol requires $\Omega(n^3)$ work for n processes to each successfully update one location once.*

Proof. Consider an adversary that simply activates every instruction as soon as it becomes ready, and assume that n instructions are ready at the start. Recall that an update attempt can only succeed if there was no successful CAS by any other process between that update attempt's read and CAS instructions. Furthermore, all CAS instructions conflict with each other, and for a CAS attempt to be successful, there have to be no other successful CAS instructions while it waits in the active set to be executed.

We partition the execution into *rounds*. Each round has exactly one successful CAS in it (the ith successful CAS is in round R_i), and ends immediately after this CAS. The execution ends after round R_n. Note that R_1 is short; all processes start with their read instructions, which takes one time step since reads do not conflict with each other, and in the next time step there is a successful CAS. However, after R_1, all processes that have not yet terminated will have a failed CAS instruction in each round. In particular, every round R_i for $i \geq 2$ has at least $n - i$ failed CAS attempts in it.

We denote by $Load(R_i)$ the minimum number of instructions in the active set at any time step in R_i, and by $Length(R_i)$ the total number of time steps in round R_i. Note that for each round R_i, $Load(R_i) \geq n - i$. This is because, in the naïve update attempt policy, no time is spent in the idle phase, and by assumption in this proof, the adversary immediately activates each ready instruction. So each process immediately returns to the active set with a new instruction after leaving the active set. Furthermore, as argued above, each round R_i has at least $n - i$ failed CAS attempts, and since all CAS attempts conflict, this means that $Length(R_i) \geq n - i$.

Recall that the work of an execution is equal to the sum over all time steps of the size of the active set. Thus, we sum over all rounds and all time steps in them, to get

$$W \geq \sum_{i=1}^{n} Load(R_i) \cdot Length(R_i) \geq \sum_{i=1}^{n} (n-i)^2 > \sum_{i=1}^{n/2} \left(\frac{n}{2}\right)^2 \in \Omega(n^3). \qquad \square$$

2.3.1 Analyzing Exponential Delay

Recall that delays executed by process p during the protocol translate to time steps p spends in the idle set before becoming ready.

The Lower Bound. To show a lower bound of $\Omega(n^2 \log n)$ on the running time of an update loop using the exponential delay protocol, we simply need to present a strategy for the adversary that forces the algorithm to take that long.

Theorem 2.3.3. *The standard exponential delay algorithm takes $\Omega(n^2 \log n)$ work for n processes to each successfully update one location once.*

Proof. Consider an adversary that simply places every operation in the active set as soon as it becomes ready, and assume that n operations are ready at the start. Recall that work is defined as the sum over all time steps of the length of the active set.

For every process p, let M_r be p's maxDelay and D_r be its actual delay after its rth update attempt. Note that the adversary's strategy is to activate p immediately after D_r steps. However, regardless of D_r, in this proof we only start charging work for p's presence in the active set M_r steps after its rth update attempt, or we charge zero if the process already finished after M_r steps. Thus, we actually charge for less work than is being done in reality.

Note that at time step t, there will be $n - d_t$ processes in the active set, where d_t is the number of processes that are done by time t or are delayed at time t. Then note that any single process p can only do at most one update attempt per n time steps. This is because all update attempts contain a CAS, and thus conflict with each other under our model. Furthermore, after p executes one update attempt, it must delay for maxDelay time steps, and then wait behind every other instruction in the active set. The active set starts with n instructions at the beginning of the execution. This also means that successful CAS instructions happen at most once every n time steps.

Every time a process executes an update attempt, assuming it does not terminate, it doubles its maxDelay. As noted above, for every process p, this happens about once every n time steps. As delays grow, there are proportionally many processes that are delayed at any given time t. We can model this as $d_t \leq 2^{\lceil t/n \rceil + 1}$. We now calculate the amount of work W this execution will take.

$$\begin{aligned} W &= \sum_t |A_t| = \sum_t \max\{n - d_t, 0\} \\ &\geq \sum_t \max\{n - 2^{\lceil t/n \rceil + 1}, 0\} \\ &= n \cdot (n-4) + n \cdot (n-8) + \ldots + n \cdot (n-n) \\ &= \Omega(n^2 \log n). \end{aligned}$$

□

The Upper Bound. Recall that for the purpose of this analysis, we use Assumption 2.3.1. Given this assumption, we show that once every process's max delay reaches $12n$, the algorithm terminates within $O(n \log n)$ work w.h.p.. This is because sufficiently many time slots are empty to allow quick progress. Thus, the adversary can only expand the work while not all processes' maxDelays have reached at least $12n$. We show that the adversary cannot prevent this from happening for more than $O(n^2 \log n)$ work w.h.p..

Lemma 2.3.4. *Consider n processes attempting to update the same memory location, using the exponential delay algorithm, and assume all of them have a max delay of at least S by some time t. Then in any segment of length S starting after time t in the execution, there will be at most $2n$ update attempts in expectation and w.h.p..*

Proof. For each process p, let $R(p, k)$ be the event that process p was ready with a new update attempt k times during the segment.

Then the probability that p will attempt k updates in the segment is bounded by:

$$P[R(p,k)] \leq \prod_{j=1}^{k} \frac{S}{2^{j-1} \cdot S} = \prod_{j=1}^{k} \frac{1}{2^{j-1}} = \frac{1}{2^{k(k-1)/2}}$$

The above calculation assumes that every time p attempts to execute and then restarts its delay, it gets to start over at the beginning of the segment. This assumption is clearly too strong, and means that in reality the probability of attempting k updates within one segment decays even faster as k grows. However, for our purposes, this bound suffices. Furthermore, note that the probability that any single process p is ready at least $2 \cdot (\sqrt{\log n} + 1)$ times is:

$$P[R(p, 2 \cdot (\sqrt{\log n} + 1))] \leq \frac{1}{2^{(\sqrt{\log n}+1)\sqrt{\log n}}} < \frac{1}{n}.$$

Thus, no process will be ready in any one segment more than $O(\sqrt{\log n})$ times w.h.p.. Let N be the number of update attempts by all the processes during the segment. We can calculate the expected number of update attempts in the segment as follows:

$$E[N] < \sum_{k=1}^{S} n \cdot P[R_i(p,k)] = n \sum_{k=1}^{S} \frac{1}{2^{k(k-1)/2}} < \frac{7}{4}n$$

Using Hoeffding's inequality, we can similarly bound the number of update attempts in the segment with high probability:

$$\begin{aligned} P(X - E[X] \geq \delta) &\leq \exp(-\frac{2n^2\delta^2}{\sum_{i=1}^{n}(\max_i - \min_i)^2}) \\ P(X \geq \frac{15}{8}) &\leq \exp(-\frac{n^2}{32n(\sqrt{\log n})^2}) \\ &= \exp(-\frac{n}{32\log n}) \end{aligned}$$

Therefore, the number of update attempts ready by processes with max delay at least S during any segment of length S is bounded by $\frac{15}{8}n < 2n$ in expectation and with high probability. □

Lemma 2.3.5. *Consider n processes attempting to update the same memory location, using the exponential delay algorithm. Within $O(n \log n)$ time steps and $O(n^2 \log n)$ work, all processes will have a max delay of at least $12n$, with high probability.*

Proof. We partition the set of processes P into three subsets – small-delay, large-delay and done – according to the processes' state in the execution, and update these sets as the execution proceeds.

For a given time step t, let S_t be the set of *small-delay* processes, whose max delay is small, i.e. maxDelay $< 12n$. Similarly, let L_t be the set of *large-delay* processes whose, max delay is large, i.e. maxDelay $\geq 12n$, and let D_t be the set of processes that are *done* (i.e., have already

CASed successfully and terminated by time t). Note that $S_0 = P$, $L_0 = D_0 = \emptyset$, that is, at the beginning of the execution, all processes have a small max delay. Furthermore, by definition, the execution ends on the first time step t such that $D_t = P$ and $S_t = L_t = \emptyset$.

We want to bound the amount of work in the execution until the first time step t such that $L_t \cup D_t = P$. To do so, we need to show that small-delay processes make progress regularly. That is, we need to show that the adversary cannot prevent small-delay processes from raising their delays by keeping them in the ready set for long. By Assumption 2.3.1, if only small-delay processes are in the ready set R_t at some time t, then the adversary must activate at least one of them. Thus, we show that in any execution, a constant fraction of the time steps have no large-delay processes ready.

We split up the execution into segments of length $12n$ time steps each. By Lemma 2.3.4, in each such segment S_t that starts at time t, there are at most $2|L_t|$ update attempts by processes in L_t with high probability. Thus, even if $|L_t| = \Theta(n)$, in every segment of size $12n$, there is a constant fraction of time steps (at least $10n$ with high probability) in which no process in L_t is ready. Furthermore, note that since their max delays are $< 12n$, every process in S_t must be ready at least once in every segment. Recall that the adversary must make at least one instruction active in every step in which there is at least one instruction that is ready. Thus, allowing for processes in S_t to 'miss' the empty time steps in one segment, we have the following progress guarantee: at least once every two segments of length $12n$, at least n instructions of processes with small delays will be executed.

Note that every process needs to execute $\log 12n$ instructions in order to have a large delay. Therefore, for all processes to arrive at a large delay, we need $n \log 12n$ executions of instructions that belong to processes with a small delay. By the above progress guarantee, we know that every $24n$ time steps, at least n such instructions are executed. Thus, in at most $24n \log 12n$ time steps, all processes will reach a delay of at least $12n$, with high probability. Note that in every time step t, there can be at most n instructions in the active set, A_t. Thus, the total work for this part of the execution is bounded by $W = \sum_t^{24n \log 12n} |A_t| = O(n^2 \log n)$ with high probability. □

Lemma 2.3.6. *Consider n processes attempting to update the same memory location, using the exponential delay algorithm. If the max delay of every process is at least $12n$, then the algorithm will terminate within $O(n \log n)$ additional work with high probability.*

Proof. We can apply the analysis of linear probing in hash tables [185] to this problem. Every process has to find its own time slot in which to execute its instruction, and if it collides with another process on some time slot, it will remain in the active set, and try again in the next one.

We think of the update attempts of the processes as elements being inserted into a hash table, and the time slots as the bins of the hash table. If a process's instruction collides with another's then it will 'probe' the next time slot, until it finds an empty slot in which to execute. Counting the number of probes that each instruction does gives us exactly the amount of work done in the execution.

For the purpose of this analysis, we assume that an update attempt succeeds if and only if neither its read nor its CAS collide with any other instruction. If either one of them does, we assume that this process will have to try again. Furthermore, since each update attempt involves two instructions, we put two time slots in a bin; that is, we assume the number of bins is half of

the minimum max delay of the processes (in this case, we know all processes have max delay $\geq 12n$). Thus, we effectively double the load factor in our analysis to account for the two instructions of the update attempts.

Thus, if we have $k \leq n$ update attempts, and all of their processes have max delay at least m, then the maximum amount of work for one update attempt of any process is at most the number of probes an element would require to be inserted into a hash table with load factor $\alpha = 2k/m$.

We start with n processes participating, all of them having a max delay of at least $12n$. By Lemma 2.3.4, we know that there are at most $2n$ update attempts in $12n$ time steps, with high probability. Thus, our load factor is $\alpha = (2 \cdot 2n)/12n = 1/3$. Let X_i be the amount of work that process p_i does for one update attempt. Let $X = \frac{1}{n}\sum_i E[X_i]$. It is well known [185] that in this case, the expected number of probes an insertion does in linear probing is at most

$$E[X] = \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right) = \frac{1}{2}\left(1 + \frac{9}{4}\right) = \frac{13}{8}.$$

Thus, by Markov's inequality,

$$P[X \geq \frac{15}{8}] \leq \frac{8E[X]}{15} = \frac{13}{15}.$$

Thus, $X < \frac{15}{8}$ with probability at least $\frac{2}{15}$. Note that the amount of work per process is integral and at least one (when it experiences no collisions). Therefore, a simple averaging argument shows that at least $\frac{1}{8}$ of the processes cost only 1, and thus experience no collisions, with probability at least $\frac{2}{15}$. Any process that experiences no collisions terminates in that step. Therefore, with constant probability, a constant fraction of the processes terminate out of every n update attempts.

We can imagine repeatedly having rounds in which the remaining processes are inserted into a hash table of size. Note that the expected cost for n processes to do one update attempt is at most $\frac{16n}{13}$. Furthermore, note that this cost is in reality even better, since process delays increase and the number of inserted instructions decreases, both causing less collisions as the load factor α decreases. However, even if we do not take the decreasing load factor into account, a constant factor of the processes terminate in every such 'round'. Note that since we do not account for the improvements from the previous rounds when calculating the expected cost of round i, the rounds are independent. Thus, we can apply the Chernoff bound to show that within $O(\log n)$ rounds, all processes successfully update, costing a total of $O(n \log n)$ work with high probability. □

The lemmas above immediately lead to the following theorem.

Theorem 2.3.7. *The standard exponential delay algorithm takes $O(n^2 \log n)$ work with high probability for n processes to each successfully update the same location once.*

2.4 A New Approach: Adaptive Probability

We present a new protocol that beats the standard exponential delay protocol under our model. The pseudocode for our algorithm is given in Algorithm 2.3. We say that every iteration of the while loop executed by process p is an update attempt, or a *step* by p.

Each process maintains a probability $prob$, which it updates in every step. All processes start with $prob = 1$. Each process also keeps track of the value val that was returned by its most recent read. On each step, p tries to CAS with probability $prob$ into location $x$and if successful it is done. Otherwise it re-reads the value at x and if changed (indicating contention) it halves its probability, and if not changed it doubles its probability. Intuitively, this protocol lets processes receive more frequent feedback on the state of the system than exponential delay (by reading the contended memory), and hence processes can adjust to different levels of contention faster. Furthermore, 'backing-on' allows processes to recover more quickly from previous high loads. Since reads do not conflict with each other in our model, reads between CASes are coalesced into a single time step, and do not significantly affect work, as our analysis will show. In particular, we show that for n processes to successfully update a single memory location following our protocol, it takes $O(n^2)$ work in expectation and with high probability.

```
1  update(x,modify){
2    currentVal = Read(x);
3    prob = 1;
4    while true { //one iteration is an update attempt
5      heads = flip(prob);
6      if (heads) {
7        val = modify(currentVal);
8        if (CAS(x,currentVal, val)){ return; }
9        newVal = Read(x);
10       if (newVal == currentVal){
11         prob = max(1, 2·prob);
12       } else {
13         prob = prob/2;
14         currentVal = newVal;  } } } }
```

Figure 2.3: Adaptive Probability

2.4.1 Analysis

The most important factor for understanding the behaviour of our algorithm is the CAS-probabilities of the processes. These probabilities determine the expected number of reads and CAS attempts each process does during the execution, and thus they greatly influence the overall work.

We first analyze an execution for a single process in isolation, and then consider the progress of the execution as a whole, when all processes participate. Consider any process $p \in P$ as it executes Algorithm 2.3. Let E be any legal execution, and $E|p$ be the execution restricted to only p's instructions. For the rest of this discussion, and for the proof of Lemma 2.4.1, when we discuss numbered steps, we mean p's steps in $E|p$. Recall that a step is defined to be a single iteration of the while loop (or a single update attempt), and thus every step may contain more than one instruction. For every process $p \in P$, we define p's *state* at step t, $s(t)$, to be i if p's probability at step t is 2^{-i}. Note that p's state is never the same in two consecutive steps. Furthermore, as long as p doesn't terminate, p's state can only change by one in every step. That is, if $s(t) = i$, then $s(t+1) = i+1$ or $s(t+1) = i-1$. We can model a process p's states over

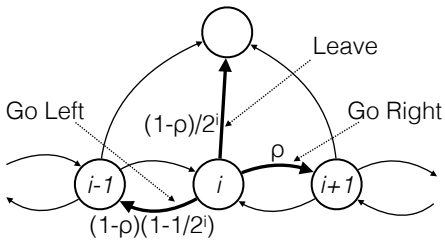


Figure 2.4: Simple state transition system for one process's execution. The transitions p can make are labeled with their name and their transition probability. The probability ρ is determined by the adversary.

the execution using a simple state transition system (Figure 2.4). In addition to these states, we need one state in the chain to represent p terminating.

The transition probabilities between p's probability states depend on the other processes in the system. Note that in our algorithm, p decreases its probability in step t, or transitions to the right, if and only if there has been a successful update by some other process between p's read and its CAS. Otherwise, p's state transition will depend on its own CAS-probability—if it CASes, its successfully updates and terminates, and otherwise it goes left. Thus, p's state transition depends on whether other processes successfully update. This is heavily controlled by the adversary.

Recall that we assume an oblivious adversary. That is, the adversary cannot base its decision of which processes to activate on their local values. In particular, that means that we do not allow the adversary to know the result of a process's coin flips, and thus whether it will read or CAS in the next step. However, we do allow the adversary to observe the instruction being done once that instruction is in the active set, and thus keep track of each process's history to know its state at every time step.

Therefore, the adversary can decide which instructions to activate given the probability states of their process. In particular, when considering a single process p's state transition, the adversary can decide how many processes it activates between two consecutive steps of p and how likely those processes are to successfully update. Thus, we can abstract the adversary's power by saying it can pick any probability ρ for how likely the state will go right, i.e., conflict with another process and decrease its probability. Note that in reality, if all participating processes are using the same protocol, the adversary will be more restricted than this, however despite the power we are giving the adversary this abstraction still allows us to prove the first important lemma of our analysis.

Lemma 2.4.1. *Every process executes at most 4 CAS attempts in expectation before terminating.*

For this Lemma, we will focus on any one process $p \in P$, and track its movement in the state transition system during a given execution E. In the rest of this section, unless otherwise indicated, all analysis refers to process p and execution E of Algorithm 2.3.

We denote by $R(i)$ the number of 'right' state transitions of p from state i to state $i+1$ in E, and by $L(i)$ the number of 'left' state transitions of p from i to $i-1$ in E. We first make an observation about these state transitions.

Observation 2.4.2. *For every state i, $L(i+1) \leq R(i)+1$.*

That is, the number of transitions from $i+1$ to i is at most the number of transitions from

i to $i+1$, plus 1. This can be easily seen by considering, for every time p transitions from i to $i+1$, how it got to state i. p starts at state 0. Thus, for every state $i > 0$, the first time p reaches state $i+1$ is by going right from state i. However, for every subsequent time p transitions right from i to $i+1$, it must have, at some point, gone left from $i+1$ to i. Thus, the number of times p transitions from i to $i+1$ is at most one more than the number of times it transitions from $i+1$ to i.

Proof of Lemma 2.4.1. Consider execution $E|p$; that is, the execution E restricted to just the instructions of p, and let C_E be the number of CAS attempts in this execution. We partition $E|p$ into two buckets, and analyze each separately; for every state i, we place the first time p reaches i into the first bucket, denoted *Init*, and place every remaining instruction in the execution into the second bucket, called *Main*. Thus, $C_E = C_I + C_M$, where C_I and C_M correspond to the cost (number of CAS attempts) of *Init* and *Main* respectively.

Note that *Init* corresponds to an execution in which p only goes down in probability. In every state i it reaches, it has probability $\frac{1}{2^i}$ of attempting a CAS, and thus expected cost of $\frac{1}{2^i}$ for that state. Thus, we can easily calculate its expected cost.

$$E[C_I] = \sum_{i=0}^{\infty} 2^{-i} \leq 2.$$

We now consider the *Main* bucket of the execution $E|p$. There are two random variables that affect the steps that p takes. The adversary can affect whether p goes right or not, by picking a value for ρ (see Figure 2.4). However, if p does not go right, then only p's coin flip determines whether it will go left or leave (this is independent of any choice of the adversary). We will charge the right transitions, that the adversary can affect, against the left-or-leave transition that the adversary cannot. We can do this since for every left transition from a state i in the main bucket, there is at most one corresponding right transition from i when it goes back to the right through state i (there might be none if it terminates at a position to the left).

We define D to be the sequence of times t in *Main* in which p does not go right, and use j to indicate indices into this sequence (to avoid confusion with i for state and t for position in $E|p$). Let $C_{M,j}$ to be the random variable for the remaining cost of *Main*, after the jth step in D, but not including the final successful CAS. If just before the jth step in D, the process is at state i, there is a $c_j = \frac{1}{2^i}$ probability that the process will CAS and terminate. Thus, there is a $1 - c_j$ probability that the execution will continue. We let R_j be the random variable indicating that the future right transition assign to j does a CAS. The expected cost (probability) of CASing when going right from state i is $\frac{1}{2^i}$, but since there might not be a future right transition, $E[R_j] \leq c_j$. We now have the following recurrence for the expectation of $C_{M,j}$:

$$\begin{aligned} E[C_{M,j}] &= E[R_j] + (1 - c_j)\, E[C_{M,j+1}] \\ &\leq c_j + (1 - c_j)\, E[C_{M,j+1}] \end{aligned}$$

The c_j's represent the results of independent coin flips, and therefore the expectations can be

multiplied. We therefore have:

$$\begin{aligned}
E[C_{M,1}] &\leq c_1 + (1-c_1)\cdot E[C_{M,2}] \\
&= c_1 + (1-c_1)(c_2 + (1-c_2)(c_3 + (1-c_3)(\ldots))) \\
&= 1 - (1-c_1) + (1-c_1)\cdot c_2 + (1-c_1)(1-c_2)\cdot c_3 + \ldots \\
&= 1 - (1-c_1)(1-c_2) + (1-c_1)(1-c_2)\cdot c_3 + \ldots \\
&= 1 - (1-c_1)(1-c_2)(1-c_3)\ldots \\
&\leq 1
\end{aligned}$$

Recall that in addition to this cost, there is exactly one successful CAS that causes p to terminate, so $C_M = C_{M,1} + 1$. Adding this to the cost of the two buckets of the execution, we can bound the total expected cost of $E|p$ by $E[C_E] = E[C_I + C_M] \leq 2 + 1 + 1 = 4$. □

We now consider the total number of steps that p does in E. Recall that a step, or an update attempt, is defined as an iteration of the loop in Algorithm 2.3, even if the process doesn't execute a CAS. We first show that the maximum number of steps of p depends on the number of successful updates by other processes. For this purpose, we stop considering $E|p$ in isolation, and instead look at the execution E as a whole.

Lemma 2.4.3 (Throughput lemma)**.** *For any process p and state i, let t and t' be two times in E at which p executes a read instruction that causes it to transition to state i. Then the number of steps p takes between t and t' is at most $2k$, where k is the number of successful updates in the execution between t and t'.*

Proof. Let $E' \subseteq E$ be a sub-execution where the first and last instructions in E' are by process p, and both bring p to state i. Let k be the number of successful updates that occur during E'.

We say that p *observes* a successful update w if p decreased its probability due to the value written by w. We note that for p to return to a state i, it has to raise its probability exactly as many times as it decreases it, when counting starting at the last time it was at state i. Note that p can only decrease its probability if it observes a new successful update. Since it must change its probability on every step, p raises its probability on every step where it does not observe a new update. Thus, one observed update's effect gets 'canceled out' by one update attempt of p in which it did not observe any successful updates. Clearly, p can observe at most k successful updates during E'. Thus, it can make at most k update attempts raising its probability, for a total of at most $2k$ steps during E'. □

Corollary 2.4.4. *Every process makes at most $2n$ update attempts during an execution of Algorithm 2.3, where n is the number of successful updates in the execution.*

Proof. Consider any process $p \in P$ and recall that every process starts the execution at state $i = 0$. At this state, p has probability 1 of attempting a CAS. If, at any time, p is in state 0, and no successful update happens before its next step, then p succeeds and terminates. Note that there are at most $n - 1$ successful updates by other processes during p's execution, since each process only updates once. By Lemma 2.4.3, if p takes $2n - 2$ steps without terminating, then p is back at state 0, and all other processes have executed successful updates. Thus, p's next step is a CAS that succeeds, since all other processes have terminated. Therefore, p can make at most $2n$ update attempts before it succeeds. □

We are now ready to analyze the entire execution.

Theorem 2.4.5. *Algorithm 2.3 takes $O(n^2)$ work in expectation for n processes to each successfully update the same location once.*

Proof. Note that by Corollary 2.4.4, every process executing Algorithm 2.3 does at most $O(n)$ instructions until it terminates. Thus, in the entire execution, there are $O(n^2)$ instructions. Furthermore, by Lemma 2.4.1, only $O(n)$ of these are CAS attempts in expectation. The rest must be read instructions.

Recall that $W = \sum_t |A_t|$, and that for all times t, $|A_t| \leq n$. Furthermore, recall that CAS instructions conflict with every other instruction, but read instructions do not conflict with each other. Thus, a CAS instruction w executed at time t can cause every instruction $i \in A_t$ such that $i <_A w$ to be delayed one time step. There are at most $O(n)$ such delayed instructions per CAS instruction. In addition, any read instruction will be active for exactly one time step if there is no CAS instruction in front of it. Thus, we have: $W = O(n) \cdot O(n) + O(n^2) = O(n^2)$. □

We can extend our results to be with high probability. For this, the key is to show that there is a linear number of CAS attempts with high probability in every execution, and then the rest of the results carry through.

Lemma 2.4.6. *There are $O(n)$ CAS attempts in any execution with high probability.*

Proof. We first show that, with high probability, no process attempts more than $O(\log n)$ CAS instructions. To do this, we define an indicator variable $X_i = 1$ if p executes a CAS instruction in its ith update attempt, and $X_i = 0$ otherwise. Note that X_i is independent from X_j for all $i \neq j$. Furthermore, let $X = \sum_i X_i$. In Lemma 2.4.1, we show that $E[X] \leq 4$. We can now use the Chernoff inequality to bound the probability of a single process attempting more than $c \log n$ CAS instructions for any constant c.

$$P[X > (1 + c \log n) \cdot 4] \leq e^{\frac{-4c \log n}{3}} = \frac{1}{n^{4c/3}}$$

Thus, every process attempts at most $O(\log n)$ CAS instructions w.h.p. We can now use this bound on the number of CAS attempts per process when bounding the probability of getting more than a linear number of CAS attempts in the system in any execution.

We can now number the processes, and define, for each process p_i, $Y_i = w_i$ where w_i is the number of CAS attempts p_i executes in E. We assume that for every i, $Y_i \in [0, c \log n]$ for some constant c. Let $\bar{Y} = 1/n \sum_{i=1}^{n} Y_i$. Then by Lemma 2.4.1, $E[\bar{Y}] = 4$. Plugging this into Hoeffding's inequality, we get

$$P[\bar{Y} \geq 5] \leq e^{\frac{-2n}{c \log^2 n}}$$

Therefore, with high probability, there are at most $5n$ CAS attempts in the execution E. □

We thus have the following theorem.

Theorem 2.4.7. *Algorithm 2.3 takes $O(n^2)$ work in expectation and with high probability for n processes to each successfully update the same location once.*

2.5 Related Work: Other Contention Models

Contention has been the subject of a substantial amount of previous work. Many researchers noted its effect on the performance of algorithms in shared memory and used backoff to mitigate its cost [20, 135, 148, 227]. Anderson [20] used exponential delay to improve the performance of spin locks. Herlihy [148] used a very similar exponential delay protocol in read-modify-write loops of lock-free and wait-free algorithms. However, all of these studies, while showing empirical evidence of the effectiveness of exponential delay, do not provide any theoretical analysis.

Studying the effect of contention in theory requires a good model to account for its cost. Gibbons, Matias and Ramachandran first introduced contention into PRAM models [131, 132]. They observed that the CRCW PRAM is not realistic, and instead defined the QRQW PRAM. This model allows for concurrent reads and writes, but queues them up at every memory location, such that the cost incurred is proportional to the number of such operations on location. Their model also allows processes to pipeline instructions. That is, processes are allowed to have multiple instructions enqueued on the memory at the same time. Instructions incur a delay at least as large as the length of the queue at the time they join.

Dwork, Herlihy and Waarts [110] defined memory *stalls* to account for contention. In their model, each atomic instruction on a base object has an *invocation* and a *response*. An instruction experiences a stall if, between its invocation and response, another instruction on the same location receives its response. Thus, if many operations access the same location concurrently, they could incur many stalls, capturing the effect of contention. In this model, the adversary is given more power than in ours. In particular, we can imagine our model's 'activation' of an instruction as its invocation. Our model then allows the adversary to control the order of invocations of the instructions, but not their responses. Furthermore, Dwork *et al.*'s model does not distinguish between different types of instructions—all are assumed to conflict if they are on the same location. Thus, our model is similar to a version of Dwork *et al.*'s model that restricts the allowed executions according to instructions' invocation order and only considers conflicts with modifying instructions. However, they still cannot deal with exponential delay since they have no direct measure of time.

Fich, Hendler and Shavit [113] defined a different version of memory stalls, based on Dwork et al.'s work. Their model distinguishes between instructions, or *events*, that may modify the memory and ones that may not. An instruction is only slowed down by *modifying* instructions done by different processes on the same memory location. They show a lower bound on the number of stalls incurred by a single instruction in any implementation of a large class of objects. In our model, we consider the same types of conlicts as Fich *et al.* do. However, Fich *et al.* use the same fully adversarial model as in Dwork *et al.*'s work. Thus, our model allows only a subset of the executions allowed by theirs. However, even given the same execution, the two models do not assign it the same cost. In particular, in Fich *et al.*'s model, the only instructions that stall a given instruction i are the modifying instructions *immediately* preceding it. Thus, if there is at least one read instruction between any constant number of modifying instructions, each will only suffer a constant number of stalls. The same is not true for our model.

Further work in this area was done by Hendler and Kutten [146]. Their work introduces a model that restricts the power of the advsersary. They define the notion of *k-synchrony*, which, intuitively, states that no process is allowed to be more than a k factor faster than any other process.

They consider a model which assigns a cost to a given linearized execution by considering the dependencies between the instructions. While in their analysis they assume that all instructions on the same location contend, they discuss the possibility of treating read instructions differently than writes.

Atalar *et al.* [26] also address conflicts in shared memory and their effect on algorithm performance. They consider a class of lock-free algorithms whose operations do some local work, and then repeat a 'Read-CAS loop' (similar to our definition of an update attempt) until they succeed. Their proposed model divides conflicts into two types; hardware and logical conflicts. This division is reminiscent of our model's ready vs. active instructions. However, the two classifications do not address the same sources of delays.

Contention has been studied in many settings other than shared memory as well. Bar-Yehuda, Goldreich and Itai [35] applied exponential backoff to the problem of broadcasting a message over multi-hop radio networks. In this setting, there are synchronous rounds in which nodes on a graph may transmit a message to all of their neighbors. A node gets a message in a given round if and only if exactly one of its neighbors transmitted in that round. They show that using exponential backoff, broadcast can be solved exponentially faster than any deterministic broadcast algorithm. Radio networks have since been extensively studied, and many new algorithms using backoff have been developed [34, 129, 140].

Similar backoff protocols have been considered for communication channels. Bender *et al.* [52] consider a simple channel with adversarial disruptions. They show that using a back-off/back-on approach, in which probability of transmission is allowed to rise in some cases, achieves better throughput. This is similar to the approach taken by our adaptive probability protocol. An in-depth study of backoff on multiple access channels is given by Bender *et al.* in [53]. They introduce a new backoff protocol which achieves good throughput, polylog transmission attempts, and robustness to a disruptive adversary. Further work has been done on different varieties of communication channels [33, 54, 117, 118].

The models considered in these papers are similar to each other (with some variations on the initial state of the system, how collisions are handled, and the adversary's power), but differ from shared memory in several important ways. Firstly, these communication networks generally assume synchronous rounds, whereas this is not reasonable for shared memory. Furthermore, unlike shared memory, running time in networks and communication channels is measured by the number of rounds for an algorithm to complete, regardless of how many transmissions are sent per round. However, in shared memory the cost grows with the number of attempted operations. Another difference is the behavior upon collision; in shared memory, one modifying operation still goes through, and slows down the rest, whereas in other communication models, no message can be transmitted in that round. Furthermore, a single 'message' in shared memory can take multiple steps (such as an update that is implemented with two instructions). These differences make it hard to apply results from other models directly to shared memory.

Chapter 3

Measuring Scheduling Patterns under Contention

3.1 Introduction

Creating pragmatic concurrent programs is essential for making the best use of modern multicore systems. When considering what constitutes a pragmatic program, designers often aim for high throughput, but another important feature is *fairness* among the cores participating in the algorithm. Fairness is sometimes a goal in its own right, such as in multicore web servers and other applications where each individual core's responsiveness is important. Even outside of such use cases, fairness can be important as a prerequisite for performance. Parallel programs in which work is statically assigned to cores, as is routine when using POSIX Threads[1] or OpenMP[2], often have synchronization barriers, at which point the last core to complete its work is the performance bottleneck. Such programs run faster if there is fairness among cores.

A large body of work has focused on designing algorithms that are *lock-free* or have other fairness guarantees [20, 58, 149, 227, 274]. However, as already discussed earlier in this thesis, due to a lack of an understanding of memory operation scheduling, lock-free algorithms are typically designed with an *adversarial* scheduler in mind, meaning memory operations can happen in any order consistent with the memory model. While this guarantees correctness on any hardware, it leads to overly pessimistic predictions of performance and fairness.

A recent line of work aims to relax adversarial scheduling assumptions to better reflect reality [15, 16, 26, 27, 43, 150]. It is well-known that if the hardware schedule guarantees fairness properties, then algorithms can be faster, simpler, and more powerful [43, 109, 223]. However, it is unclear whether such assumptions are realistic. Thus, to understand the performance of lock-free algorithms, we must study the *scheduling of memory operations in hardware*.

Let us first consider the kinds of demands that most concurrent lock-free algorithms make on the scheduler. Many lock-free algorithms have the structure shown in Algorithm 3.1 [16, 27]. All cores run *parallel work* (line 3), that they do independently, and then synchronize in an *atomic modify* section (lines 4–7). Note that this atomic modify section is similar to the update attempt

[1] `https://ieeexplore.ieee.org/document/8277153/`
[2] `https://www.openmp.org`

```
1 success = false
2 while true{
3   parallel_work();
4   while (not success) {
5     currentVal = Read(x);
6     val = modify(currentVal);
7     success = CAS(x, currentVal, val);  } }
```

Figure 3.1: Generic lock-free algorithm (simplified)

considered in Chapter 2. In this section, a core executes a modification of location x that must not be interrupted by any other core's modification of x. Thus, the ordering, or *schedule*, of reads and CASes of x has a large impact on the fairness and performance of the algorithm. Intuitively, a good schedule has:

- *Long-term fairness:* we want each core to perform the same number of read and successful CAS instructions over any sufficiently long period of time.
- *Short-term focus:* for performance, whenever a core reads x, we want it to execute its following CAS without other cores performing any read or CAS instructions in between.

Having outlined what a good memory operation schedule looks like, we ask: what do memory operation schedules look like on modern hardware? Do practical schedules have the fairness and focus properties we want for lock-free algorithms?

Unfortunately, this is a difficult question to answer because the complexity of modern memory hierarchies makes scheduling patterns difficult to predict. Design decisions in aspects such as the cache coherence protocol and non-uniform memory access (NUMA) can have a drastic impact on the schedule. However, exactly how different designs correspond to scheduling patterns is unclear, especially when multiple features interact with one another.

For example, it is well known that the latency of a local-node cache hit is much lower than that of a remote-node cache hit [138]. This encourages the design of NUMA-aware algorithms [58, 69, 70, 210] that minimize remote-node memory accesses. However, recent work on arbitration policies in the processor-interconnect [269] shows that when most but not all memory accesses are local—which is exactly the situation for many NUMA-aware algorithms—hardware can unfairly bias the schedule towards *remote* nodes. Thus we see that a NUMA architecture can yield unexpected schedules.

3.1.1 Our Contributions

In this chapter, we provide a way to test the schedules produced by today's machines and find patterns that can be important for fairness and performance. To do so, we introduce a benchmarking tool, called *Severus*, that allows the user to specify a workload, and tracks the execution trace produced. We show how to use Severus to understand the scheduling patterns of two modern NUMA machines, and provide a plotting library that helps intuitively visualize the results.

Severus allows the user to play with several parameters of the execution, including which threads participate in a run, what locations are accessed, how much local work each thread

does, and how long each thread waits between two consecutive operations. With this flexibility, Severus can simulate the workloads that are most relevant to the user's application.

We describe Severus and use it to demonstrate the following takeaways:

- Operation schedules are not fair by default.
- Uniform random scheduling assumptions do not accurately reflect real schedules.
- The amount of local work a thread does in a lock-free algorithm, particularly the length of the *atomic modify* section, has a large but hard-to-predict impact on the algorithm's performance.
- The details of these effects are different on each platform, but these details can be revealed by tools such as Severus.

We believe that these new findings can guide both the design of new pragmatic concurrent algorithms on existing machines and the development of new memory architectures that enable faster and more fair concurrent executions.

We reach the above takeaways by studying the memory operation scheduling patterns of two NUMA machines: an AMD Opteron 6278 and an Intel Xeon CPU E7-8867 v4. These two machines exhibit different architectural designs: the Intel has four equidistant nodes and uses a hierarchical cache coherence protocol, whereas the AMD is arranged in eight nodes, with two different distances between them, and employs a flat cache coherence mechanism. We show how these design choices translate to differences in schedules. While the scheduling patterns remain mostly round-robin on AMD regardless of the cores participating in a run, on Intel, the schedule changes drastically depending on whether cores from more than one node are running. Interestingly, both machines show *higher* throughput for cores that access remote contended memory. We characterize workloads in which this phenomenon is prominent, and show how this unfairness changes as certain parameters of the program are varied.

3.2 Background and Machine Details

3.2.1 NUMA Architectures

NUMA architectures are everywhere in modern machines. Cores are organized into groups called *nodes*, and each node has cache as well as main memory (see Figure 3.2). Within a node, cores may have one or two levels of private cache, and a shared last level cache. Each core can often be split into two logical threads, called *hyperthreads*. All cores can access all shared caches and memory, through an interconnect network between the nodes. However, accesses to cache and memory in a core's own node (*local acceses*) are faster than accesses to the cache or memory of a different node (*remote accesses*).

3.2.2 Lock-Free Algorithms and Scheduling

Lock-free algorithms guarantee that progress is made in the algorithm regardless of the number of threads participating or their relative speeds. The correctness of lock-free algorithms is typically proved under an adversarial model, whereby a powerful adversary determines the schedule of

atomic operations on each location, thus controlling who succeeds and who fails at any time. The adversarial model produces robust algorithms, but lacks predictive capabilities for performance. Usually, the best performance guarantees that can be proven under an adversarial scheduler are embarrassingly pessimistic.

Thus, recent work in lock-free algorithms proposes different scheduling models, with the goal of being able to analytically predict performance. Common alternative models include that the scheduler picks the next thread uniformly at random [16, 150], or with some predetermined distribution [15]. The goal of our work is to test whether such assumptions are reasonable, and to understand what factors of modern architectures most affect the operation scheduling, and which most affect performance.

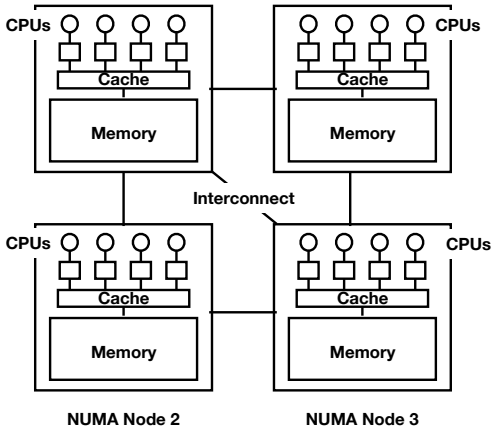


Figure 3.2: NUMA architecture with 4 NUMA nodes.

3.2.3 Machines Used

We test our benchmark on two different NUMA architectures; an Intel Xeon CPU E7-8867 v4 machine with 4 nodes and 72 cores with Quick Path Interconnect technology, and an AMD Opteron 6278 machine with 8 nodes and 32 cores, using HyperTransport. Throughout this chapter, we refer to these machines as simply *Intel* and *AMD* respectively. Both machines have a per-core L1 and L2 cache (shared among a pair of hyperthreads), and a shared L3 cache on each node. The details of the two machines are shown in Table 3.1. The Intel machine's interconnect layout is fully connected, and therefore all nodes are at the same distance from one another. However, this is not the case for the AMD machine, in which there are two different distances among the nodes. The AMD node layout and distance matrix is shown in Figure 3.3.

Both machines have an atomic compare-and-swap (CAS) instruction and an atomic fetch-and-increment (F&I) or fetch-and-add (F&A, also called xadd) instruction. A CAS instruction takes in a memory word, an old value *old*, and a new value *new*, and changes the word's value to *new* if the previous value was *old*. In this case, it returns *true*, and is said to *succeed*. Otherwise, the CAS does not change the memory word. It returns *false* and we say that it *fails*. The F&I instruction takes in a memory word and increments its value. It always returns the value of the word immediately before the increment. Both the CAS and the F&I instructions fully sequentialize accesses.

	N0	N1	N2	N3	N4	N5	N6	N7
N0	0	1	1	2	1	2	1	2
N1	1	0	2	1	1	2	2	1
N2	1	2	0	1	1	1	1	1
N3	2	1	1	0	1	1	2	2
N4	1	1	1	1	0	1	1	2
N5	2	2	1	1	1	0	2	1
N6	1	2	1	2	1	2	0	1
N7	2	1	1	2	2	1	1	0

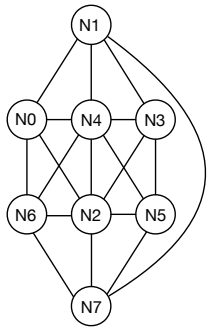


Figure 3.3: AMD node layout and distance matrix.

Table 3.1: Machine details.

SPECS	INTEL	AMD
CPU family	Xeon E7-8800	Operton 6200
Sockets	4	4
Nodes	4	8
Cores	72	32
Hyperthreading	2-way	2-way
Frequency	1200-3300 MHz	2400 MHz
L1i Cache	32k	16k
L1d Cache	32k	64k
L2 Cache	256k	2048K
L3 Cache	46080K	6144K
Coherence protocol	MESIF	MOESI

3.3 The Benchmark

Severus provides many settings to simulate the behavior of a large range of applications. For clarity, we begin by describing one simple setting, and then show ways to extend it.

At its core, Severus simply has all threads contend on updating a single memory location, either with a read-modify-CAS loop, or with an F&A. We measure throughput; how many changes to the memory location were made. To retain information about the execution, we also have a *logging* option, in which we have each thread record the values it observed on the shared location every time the thread accesses it. For the F&A case, simply recording these numbers allows us to reconstruct the order in which threads incremented the shared variable. For a CAS-based benchmark, we can control what values the threads write into the shared variable. To allow reconstruction of the execution order, we have each thread CAS in its own id and a timestamp. In this way, when threads record the values they observed, they are in effect recording which thread was the last one to modify the variable with a successful CAS. From this information, we obtain a total order of successful CASes, and a partial order on the reads and unsuccessful CAS attempts.

Severus provides parameters to modify the basic benchmark to reflect different workloads, including the following settings.

- The number of shared variables contended on.
- Which node each shared variable is allocated on.
- Which threads participate.
- For each thread, which shared variables it should access.
- Length of execution.
- Whether or not the threads should log execution information. Turning this option off helps optimize space usage.
- For CAS-based tests, delays can be injected between a read operation and the following

CAS attempt of that thread. This simulates the time it takes in real programs to calculate the new value to be written.

- Delay can be injected between two consecutive modifications of the shared variable by the same thread. This simulates programs in which threads have other work.
- Delay can also be injected between a failed CAS attempt and the thread's next read operation. This allows simulation of backoff protocols.

3.3.1 Implementation Details

When evaluating the schedule of a concurrent application, one must be very careful not to perturb the execution. Many common instructions used for logging performance, including accesses to timers, cycle counters, or memory allocated earlier in the program, can greatly affect the concurrent execution, leading to useless measurements. Thus, we take care in ensuring that our logging mechanism minimizes such accesses.

NUMA Memory and Thread Allocation. We use the Linux NUMA policy library *libnuma* to allocate memory on a specified node (both for contended locations and memory used for logging), and to specify the threads used. We pin threads to cores.

Logging. All information logged during the execution is *local*. We allocate a lot of space per thread for logging, and ensure that for each thread, this log space is in the memory of the NUMA node on which that thread is pinned. No two threads access the same log. This helps eliminate coherence cache misses that are not directly caused by the tested access pattern. Before beginning the real execution, we have each thread access its preallocated log, to avoid compulsory cache misses when it first accesses the log during its execution. Severus always records the total number of operations executed by each thread, and the total number of successful CASes per thread. This simply involves incrementing two counters, and thus never causes cache misses.

If the *logging* option is enabled, each thread also records which values it observed on the shared location when it accessed it. This logging takes much more space, since this information cannot be aggregated into one counter, and thus we keep a word per operation executed by each thread. Logging can also perturb the execution; more (uncontended) writing is done, and cache misses occur every once in while, when the size of the log written exceeds the cache size. However, since the memory of the log is accessed consecutively, prefetching helps mitigate the effect of log-caused cache misses. With this local method of logging, we process the results after the execution ends, and reconstruct the global trace from the per-process ones.

Compiler Options. To eliminate as much overhead as possible during the execution, many of the settings of a run are determined at compile time. This includes machine details, like the number of nodes and cores, and the ids of the cores on each node. The type of execution (CAS, F&A, etc.) and logging are also determined at compile time.

Delay. We implement atomic delay and parallel delay by iteratively incrementing a local volatile counter. The amount of delay given as a parameter for an execution translates to the number of iterations that are run. In the rest of the chapter, we use 'iterations' as the unit of delay used in experiments. This is done to avoid mechanisms of waiting that are too coarse grained or can perturb the execution. Therefore, given the same delay parameter, the actual amount of time that a thread waits depends on the system on which the benchmark is run (in particular, depending on the core frequency). A single unit of delay corresponds to approximately 2.2 nanoseconds on Intel and 3.5 nanoseconds on AMD (both averaged over 10 runs). We note that measuring delay in terms of iterations of local cache accesses is reasonable for simulating algorithm workloads, since it reflects the reality that different algorithms take different amounts of time on different machines.

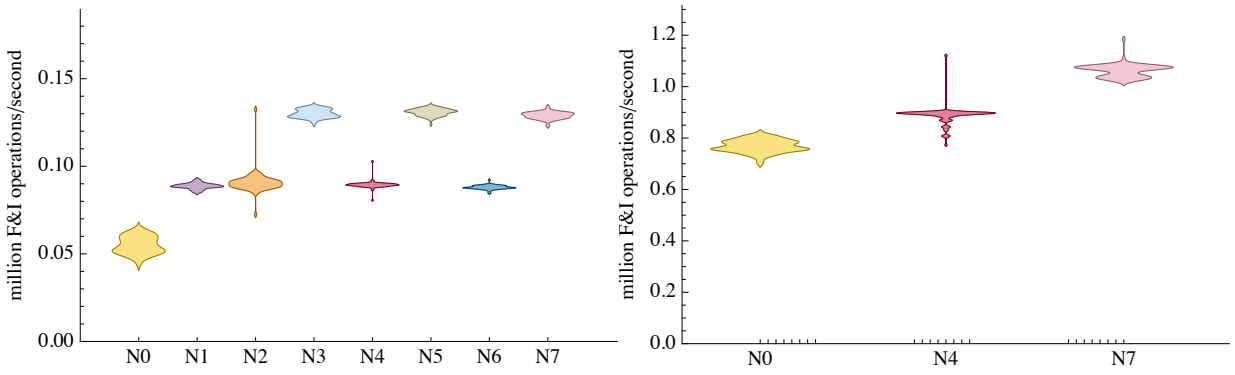
3.3.2 Experiments Shown

All tests shown in this chapter can be broadly split into two categories.

- *Sequence Experiments.* In these experiments, we take a subset of the threads (possibly all of them), and have them repeatedly increment a single location using atomic fetch-and-increment (F&I). We call the contended location the *counter*. All threads record the return value of their fetch-and-add after each operation, using the logging option. This allows us to recreate the order in which threads incremented the counter.
- *Competition Experiments.* These experiments are similar to the sequence experiments, but differ mainly in the operation used. A subset of the threads repeatedly read a location, locally modify its value, and then compare-and-swap (CAS) their new value into the same location. We call the contended location the *target*. In competition experiments, we sometimes vary other parameters, like the local modification time (which we call *atomic delay*), and the time threads wait between a successful CAS and that thread's next operation (*parallel delay*).

The competition experiments cause different scheduling patterns than the sequence ones; the read operations mean that the cache line enters the shared coherence state in addition to the modified state. Furthermore, compare-and-swaps fail if another thread has changed the value. This means that to successfully modify the location, a thread must execute two operations in a row, possibly changing its cache line's coherence state in between. The schedules produced by sequence experiments are more regular, and thus easier to analyze to obtain a high level understanding of the scheduler.

Therefore, to learn about each machine's scheduling patterns, we use sequence experiments, with the logging option turned on (Section 5.2). We show how the lessons we learn from these experiments generalize to other workloads by running competition experiments (which better reflect real-world applications), without logging, and comparing the results to the predictions made based on our learned scheduling model (Section 3.5). We also provide a script that runs the experiments described in this chapter and produces the relevant plots.



(a) All nodes participating.

(b) One node participating at a time.

Figure 3.4: AMD throughput of F&I operations. Counter allocated on Node 0. Sugfigure (a) shows all nodes participating, and Sugfigure (b) shows one node participating at a time, comparing distance 0 (Node 0), distance 1 (Node 4), and distance 2 (Node 7).

3.4 Inferring Scheduling Models

In this section, we show experiments that help determine scheduling models for the AMD and Intel machines. All the experiments in this section are *sequence experiments* (see Section 3.3). To review, in a sequence experiment, multiple cores atomically fetch-and-increment (F&I) a single memory location called the *counter*. This yields a full execution trace, namely a sequence of all the F&I operations executed by all threads, which we analyze in several ways to determine a scheduling model. Across different experiments, we vary the number of threads participating, the placement of the threads, and the NUMA node on which the counter is allocated.

A sequence experiment is a hardware stress test meant to reveal details about how it schedules memory operations. It is *not* meant to model a realistic lock-free algorithm. In particular, throughput measurements of sequence experiments should be not be interpreted as a proxy for performance of a lock-free algorithm. (In contrast, the competition experiments in Section 3.5 *are* intended to model lock-free algorithms.)

3.4.1 AMD Scheduling Model

AMD Throughput Measurements. We begin with a basic question: when all cores participate in a sequence experiment, do they achieve the same throughput? As we will see, the answer to this question is counterintuitive and will guide our more detailed analysis of the machine's scheduling model.

To answer this, we run a sequence experiment with the counter on Node 0 and simply count the number of F&I operations executed by each core. For each node, Figure 3.4a shows the distribution of throughputs among cores of that node.[3] We see that most cores within any given node have similar throughput, but different nodes have very different throughputs. We observe that the throughput is unfair:

[3]Throughout this section, all throughput distribution plots show the aggregate throughput distribution of 10 separate 10-second runs.

- Node 0, which is where the counter is allocated, has the lowest throughput;
- Node 1, Node 2, Node 4, and Node 6 have intermediate throughput; and
- Node 3, Node 5, and Node 7 have the highest throughput.

What distinguishes Node 3, Node 5, and Node 7 from the other nodes? The answer lies in Figure 3.3: they are the *farthest* from the counter on Node 0. That is, a core's throughput tends to increase with its distance from the counter. Repeating the experiment with the counter on each node confirms this.

So far, we have seen that with all cores from all nodes participating, cores on nodes farther from the counter have a throughput advantage. We now ask: does this trend still hold when nodes participate one at a time? To answer this question, we run experiments with the counter on Node 0 with cores on just a *single node* participating. Figure 3.4b shows the distribution of results for each of Node 0 (distance 0), Node 4 (distance 1), and Node 7 (distance 2) participating. Unlike the previous plots, each distribution in the plot represents a *separate configuration* in which only that node is participating. The overall throughput is higher in these configurations because of reduced contention.

Remarkably, Figure 3.4b shows that even with only a single node participating, throughput still increases with distance from the counter. Results for other nodes at distances 1 and 2 are similar to those for Node 4 and Node 7, respectively. Similar results hold when cores from any subset of nodes participate.

We have firmly established that throughput is unfair and is skewed toward cores that are farther from the counter, even when the counter's cache line remains cached on the same node. This pattern reflects the directory coherence protocol on AMD, which seems to use the interconnect even when a cache line remains on one node, likely due to the need to update its coherence state in the directory. To understand why increased interconnect use increases throughput, we need a more detailed analysis of the execution traces.

AMD Execution Trace Analysis. We now thoroughly examine the execution trace of a single sequence experiment. All cores participate, and the counter is on Node 0. We examine an execution trace excerpt of 2^{20} operations, taken from the middle of the experiment to avoid edge effects. For space reasons, we show results from just one run and focus on three nodes Node 0 (distance 0 from counter), Node 4 (distance 1), and Node 7 (distance 2). We have confirmed that the results shown are robust across several trials and other nodes at distances 1 and 2 behave similarly.

The result of a sequence experiment is an execution trace, which is an ordered list of core IDs whose ith entry is the ID of the core that executed the ith F&I operation on the counter. We can think of the trace as describing how (modify-mode access to) the counter's cache line move from core to core.

To talk about the trace and its implications for throughput, we use the following vocabulary:

- *Core visit:* a contiguous interval during which just one core performs F&I operations.[4]
- *Core visit length:* the number of F&I operations performed during a given core visit.

[4]When discussing core visits, we take "core" to specifically mean "physical core" and group its two threads together.

Table 3.2: AMD core visit length distributions.

	LENGTH 1	LENGTH 2	LENGTH ≥ 3	MEAN
Cores on Node 0	88%	9%	3%	1.147
Cores on Node 4	93%	4%	3%	1.105
Cores on Node 7	55%	34%	11%	1.585

- *Core visit distance:* the number of core visits to other cores between two visits to a given core.

A core's throughput is

- directly proportional to its average core visit length and
- inversely proportional to its average core visit distance.

For each of Node 0, Node 4, and Node 7, Table 3.2 shows the distribution of visit lengths for cores on that node. Notably, the average core visit lengths on Node 7 is roughly 40% higher than each of Node 0 and Node 4. Recall that in Figure 3.4a, Node 7 has roughly 40% higher throughput than Node 4, which in turn has higher throughput than Node 0. It thus appears that average core visit length explains the throughput difference between Node 4 and Node 7, but explaining the even lower throughput of Node 0 requires examining core visit distances.

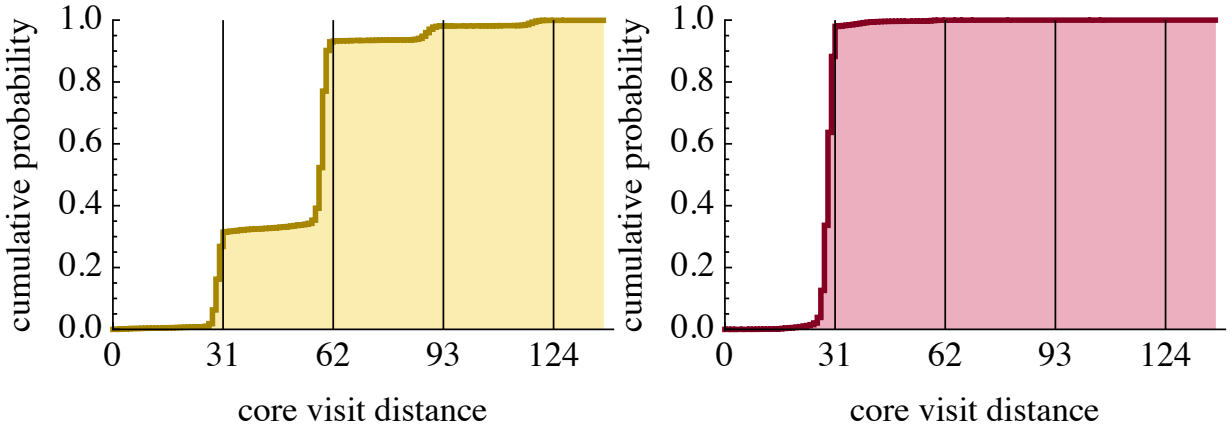
We now turn to core visit distances. Figure 3.5 shows the CDF of visit distances aggregated over all cores for Node 0 and Node 4. Due to space limitations, we omit the plot for Node 7, but it is almost identical to that of Node 4. Remarkably, nearly all core visit distances are just below multiples of 31, which is one less than the number of physical cores on the AMD machine. This suggests that core visits occur in round-robin fashion, visiting all 31 other cores between two visits to a given core, except that cores are occasionally skipped, mainly on Node 0. Given that average core visit lengths are roughly the same for Node 0 and Node 4 (see Table 3.2), their throughput difference is due mainly to the skipping of cores on Node 0.

3.4.2 Intel Scheduling Model

Intel Throughput Measurements. We begin our analysis of the Intel machine in the same way we did for AMD. We want to know whether throughput is fair among different cores, and in particular, whether the distance patterns we observed for AMD hold for Intel as well. Recall that the Intel machine has only 4 NUMA nodes, with a full interconnect that places all nodes equidistantly from one another.

Figure 3.6a shows each node's throughput distribution for a sequence experiment with all cores participating with the counter placed on Node 0. We see that, again, throughput is unfair, and cores on Node 0 have lower throughput than cores on the other three nodes. The results are analogous when the counter is allocated on Node 1, Node 2, or Node 3.

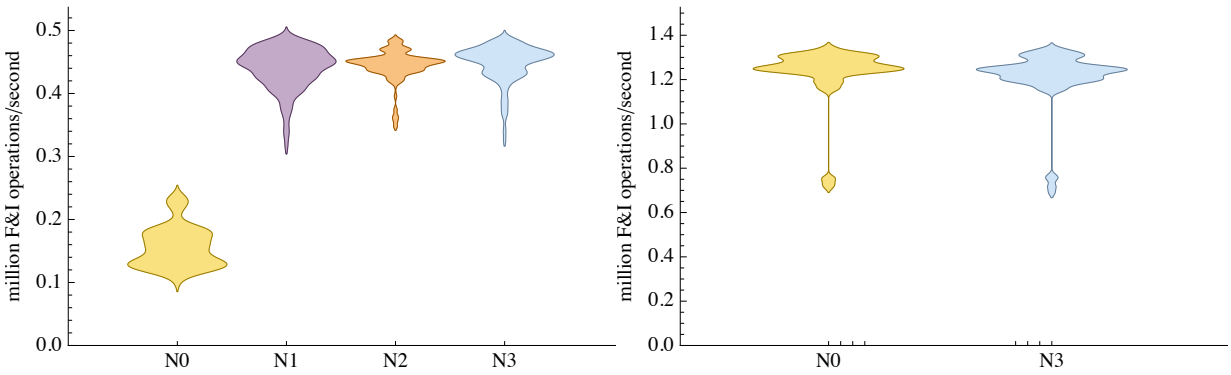
We next test whether cores close to the counter still have lower throughput when only one node participates at a time. To answer this question, we run experiments with the counter on Node 0 with cores on just a *single node* participating. Figure 3.6b shows the results for each of Node 0 and Node 3 participating, Unlike in the experiment with all nodes participating, we see



(a) Core 0 on Node 0, avg 51.6

(b) Core 16 on Node 4, avg 29.1

Figure 3.5: AMD core visit distance distributions with all nodes participating. Counter allocated on Node 0. Showing distributions for Subfigure (a) a core on Node 0 (distance 0 from counter) and Subfigure (b) a core on Node 4 (distance 1). Distributions for other distance 0 cores are similar to Subfigure (a), and likewise for distances 1 and 2 with Subfigure (b).



(a) All nodes participating.

(b) One node participating at a time.

Figure 3.6: Intel throughput of F&I operations, with counter allocated on Node 0. Subfigure (a) shows all nodes participating, and Subfigure (b) shows one node participating at a time, comparing Node 0 (distance 0 from counter) and Node 3 (distance 1).

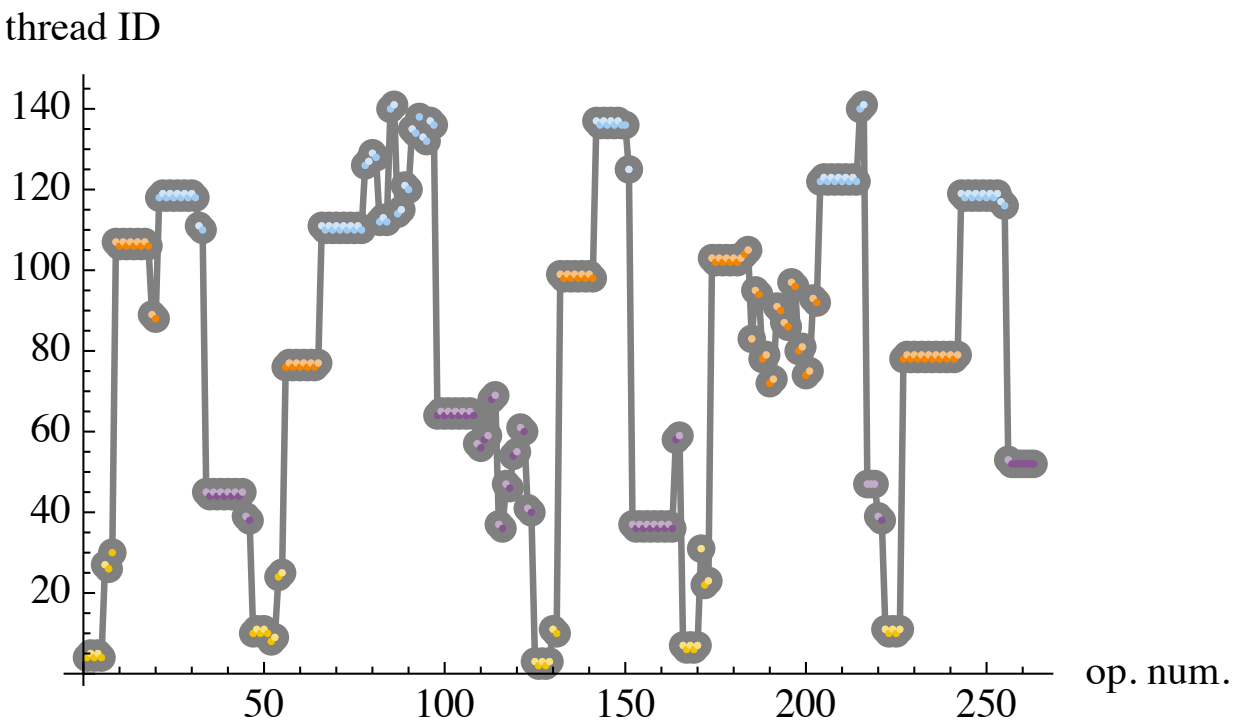


Figure 3.7: Intel execution trace of F&I operations with all nodes participating. Counter allocated on Node 0. Thread IDs are clustered by node: 0–35 on Node 0 (yellow), 36–71 on Node 1 (purple), 72–107 on Node 2 (orange), and 108–143 on Node 3 (blue). Even-odd pairs of threads (0-1, 2-3, etc.) run on the same physical core. Even thread IDs are shaded darker.

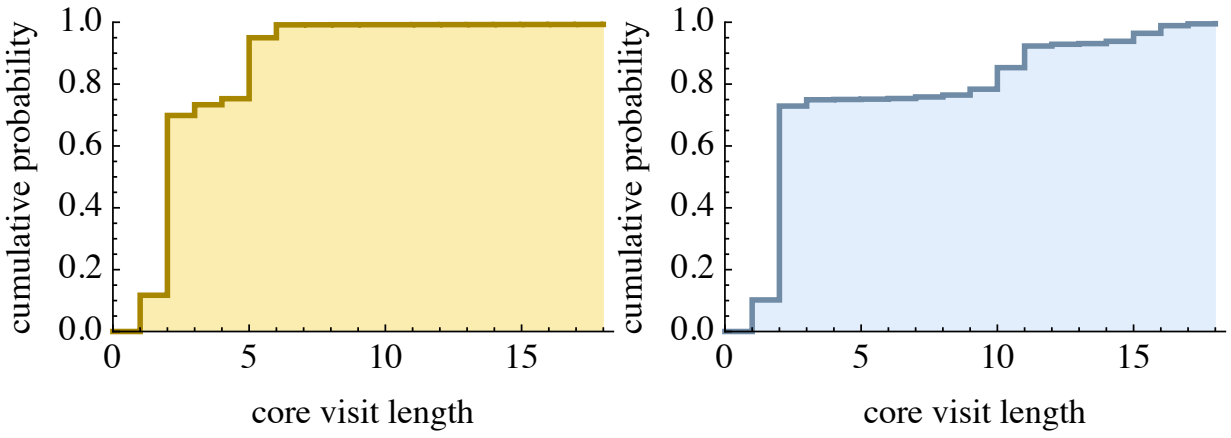
that Node 0 and Node 3 have similar throughput distributions when only one node participates at a time. The results for Node 1 and Node 2 are similar.

We have seen that with all nodes participating, Intel and AMD both exhibit core throughput increasing with distance from the counter, but the machines differ when only one node participates. This can be explained by considering the directory coherence protocol. Each node on Intel has a shared L3 cache, and the coherence protocol does not communicate updates to other nodes so long as the cache line is not in any other node's L3 cache. This means single-node runs are virtually unaffected by where the counter is allocated.

Intel Execution Trace Analysis. We now investigate the Intel execution trace in detail. Figure 3.7 shows the execution trace produced from a sequence experiment with the counter allocated on Node 0. The y-axis shows the different thread id's color-coded by node. The x-axis shows "time", measured in number of F&I operations. The line shows the counter's migration pattern across the caches of the different cores.

To discuss the execution trace, we define the following terms:

- *Core visit:* a contiguous interval during which just one core performs F&I operations (see Section 3.4.1). The *length* of a core visit is the number of F&I operations performed in it.
- *Node visit:* a contiguous interval during which cores on just one *node* perform F&I operations. The *length* of a node visit is the number of core visits it contains.



(a) Cores on Node 0, avg 2.92

(b) Cores on Node 3, avg 4.42

Figure 3.8: Intel core visit length distributions with all nodes participating. Counter allocated on Node 0. Showing aggregate distributions for a cores on Node 0 (distance 0 from counter) and b cores on Node 4 (distance 1). Distributions for Nodes 1 and 2 are similar to b.

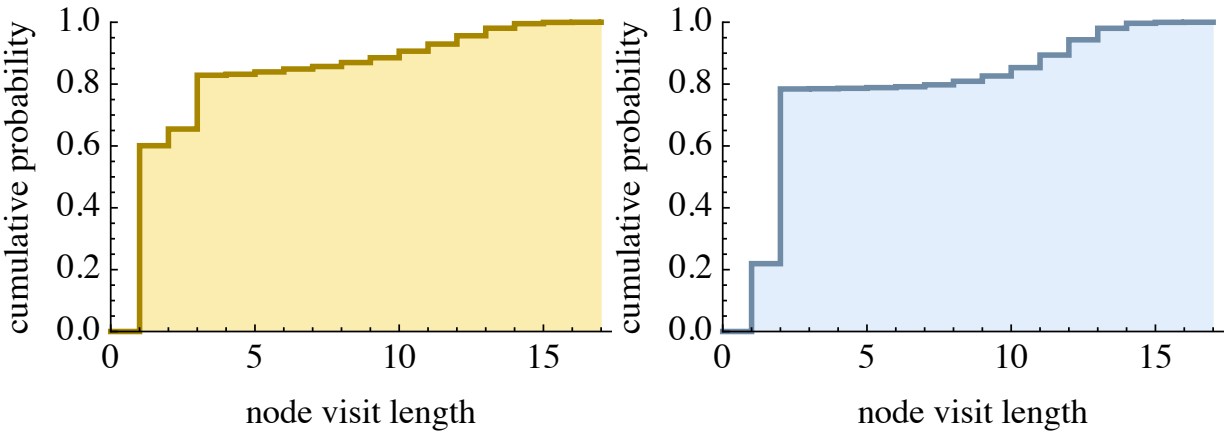
Figure 3.7 reveals unusual features of its core and node visits.

Round robin node visits: The nodes are visited in a fixed repeating order throughout Figure 3.7: 0, 2, 3, 1, ... We have confirmed that this pattern is consistent over the entire trace, though the order occasionally changes and Node 0 is occasionally skipped. We omit the detailed statistics for brevity.

Uneven core visit lengths: The first core visit of each node visit is usually relatively long. Moreover, these long core visits *only* occur as the first node visit: almost all other node visits are very short, having just one or two F&I operations. To confirm this observation, we show the CDF of the core visit length distribution for Node 0 (distance 0 from the counter) and Node 3 (distance 1) in Figure 3.8. For brevity, we omit plots for Node 1 and Node 2, which are similar to that for Node 3. The pattern is very clear for Node 3: about 70% of core visits are of length 1 or 2, but visits of length greater than 2 are likely to be at least length 10. The pattern is a bit less prominent on Node 0, where longer visits only last around 5 operations. This partially explains the difference in throughput observed between Node 0 the other nodes.

Occasional bursts: In Figure 3.7, most node visits only contain a few core visits: first a long core visit, followed by 0 to 2 more core visits. However, every once in a while, a node visit ends with many short core visits in a row. We call this occurrence a "burst" of visits. A natural question is: are bursts simply the result of noise, or they a separate phenomenon? To answer this question, we plot CDF of the node visit length distribution in Figure 3.9, again showing only Node 0 and Node 3 for brevity. The distributions make clear that there are two distinct types of node visits: those with 3 or fewer core visits, constituting about 80% of all node visits; and those with significantly more, usually at least 8, making up the other 20% of node visits. We therefore define the following terms:

- *Burst:* a node visit of length 4 or greater. For example, Figure 3.7 shows bursts for each of Nodes 1, 2, and 3.
- *Cycle:* the time between the end of one burst on a given node and the end of the next burst



(a) Node 0 **(b)** Node 3

Figure 3.9: Intel node visit length (measured in number of core visits) distributions with all nodes participating. Counter allocated on Node 0. Showing distributions for (a) Node 0 (distance 0 from counter) and (b) Node 4 (distance 1). Distributions for Nodes 1 and 2 are similar to (b).

Table 3.3: Intel number of times cores are visited per cycle.

	0 VISITS	1 VISIT	2 VISITS	$\geq$ 3 VISITS
Cores on Node 1	9%	85%	5%	$< 1\%$
Cores on Node 3	10%	85%	5%	$< 1\%$

on that node.

Interestingly, we find that in most cycles, each core is visited exactly once. This is shown in Table 3.3. This pattern, which occurs on all nodes, suggests a possible mechanism for the bursts: requests for the counter's cache line build up in a queue in each node, and each queue occasionally "flushes" if it is too full for too long.

Finally, recall from Section 3.4.2 that single-node executions produce different throughput distributions than executions that cross node boundaries. We therefore also examine the trace of a single-node execution with the counter on Node 0 and only cores on Node 0 participating. In contrast to the multiple-node trace, the single-node trace is close to uniformly random. To confirm this, we show the CDF of the core visit distance distribution in Figure 3.10. The CDF is close to that of a geometric distribution, which is what the CDF would be for a truly uniformly random schedule. This means that for analyzing algorithms for single-node executions on the Intel machine, a uniformly random scheduling model is appropriate.

3.5 Takeaways for Fairness and Focus

Recall the desirable properties a schedule should have: in the long run, we want it to be *fair*, letting each thread make the same amount of progress, but in the short term, we want the schedule to be *focused*, allowing each thread enough time to read, locally modify, and then apply its

modification on a cache line before the cache line gets invalidated.

We now go back to our original question: do memory operation schedules on modern hardware achieve long term fairness and short term focus? In the previous section, we saw some indications that the schedules might not be fair: initial throughput experiments indicated that on both machines, the node on which memory is allocated is unfairly treated, even in long runs. We saw that short-term focus might be behind this: cores on remote nodes get longer visits on average. However, recall that these experiments were *sequence* experiments, which were designed to uncover scheduling patterns but not to represent the workloads of real lock-free algorithms.

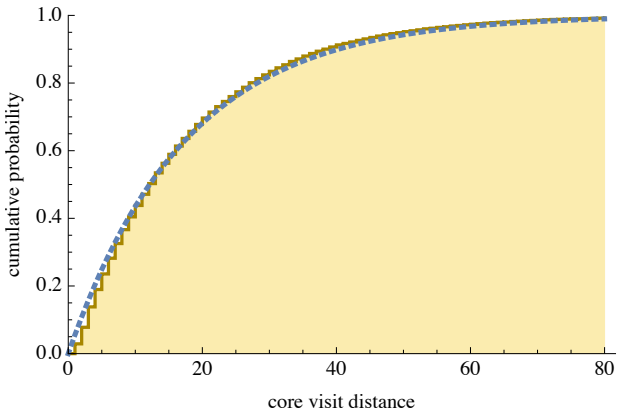


Figure 3.10: Intel core visit distance distributions with only Node 0 participating. Counter allocated on Node 0. Distribution is very close to Geometric$(1/18)$ (dashed blue line).

In this section, we thus test whether these initial findings carry over to more realistic workloads. More specifically, all the experiments in this section are *competition experiments* (see Section 3.3). To review, in a competition experiment, multiple cores attempt to read from and CAS a new value into a single memory location called the *target*. Competition experiments have two *delay* parameters.

- Between a read and the following CAS is the *atomic delay*. This simulates work in the the *atomic modify* section of a lock-free operation (Line 6 of Algorithm 3.1).
- Between each successful CAS and the following read is the *parallel delay*. This simulates the *parallel work* of a lock-free algorithm between synchronization blocks (Line 3 of Algorithm 3.1).

We simulate different lock-free workloads by varying the atomic and parallel delays. To highlight the effects of the atomic delay, the experiments in this section are conducted with a high parallel delay (set to 256 iterations in all experiments. See Section 3.3 for details on how the delay is implemented). This means that long streaks of successful read-modify-CAS operations by one thread without interruption from another thread are unlikely, even when the atomic delay is small.

All plots in this section show the results over 10 repetitions of 10 second runs. Each plot point shows the median total throughput of *successful CAS instructions* over the 10 repetitions, and error bars show the 75th and 25th percentile.

3.5.1 Fairness

To test long-term fairness on lock-free workloads, we run a set of competition experiments in which all cores on all nodes are participating. We vary the atomic delay to evaluate the fairness for lock-free algorithms with differently sized atomic modify sections. We measure the throughput of successful CAS instructions exhibited by cores on each node, and compare them to the throughput on other nodes. These tests answer the following question: when all cores run the

same code, how skewed is their throughput with respect to each other?

AMD Fairness. The results for the fairness test on the AMD machine are shown in Figure 3.11. It is clear that cores on distance 2 nodes (represented by Node 7 here) perform much better when atomic delay is low, outperforming other nodes by up to 31×, but this drops very quickly.[5] By the time atomic delay reaches 16 iterations (around 56 ns), distance 1 nodes start outperforming distance 2 nodes. However, recall that the throughput reported in Figure 3.11 shows *successful CAS* instructions. Interestingly, if we consider the number of *attempted* CAS instructions, rather than just the successful ones, the difference is less stark, with distance 2 nodes reaching a peak at an atomic delay of 5 iterations, at which point they only outperform distance 1 nodes by a factor of 2.2.[6] This indicates that at low atomic delays, distance 2 nodes succeed in a much larger fraction of their attempted CAS instructions.

The throughput reaches a steady state at around an atomic delay of 30 iterations (roughly 70 ns), but is still highly unfair. Notably, distance 1 nodes achieve the highest throughput at the steady state, outperforming the other two groups by an order of magnitude. Insight into this phenomenon can be gained by looking at the *success ratio*, or the fraction of successful CAS instructions out of the overall number attempted. For distance 1 nodes, the success ratio is around 0.04–0.05, whereas for cores in the other two node categories, it lies at around 0.005. A failed CAS is always caused by the success of another thread's CAS. In particular, a CAS by thread p will fail if p executed its read of the target between the read and the CAS of thread whose CAS was successful. Thus, the numbers indicate that most threads align their read instructions with each other, causing repeated failures for the same set of threads. The delays inherent to the cache coherence protocol on the AMD machine thus repeatedly favor these 'mid latency' (distance 1) threads over their counterparts that are farther or closer to the memory.

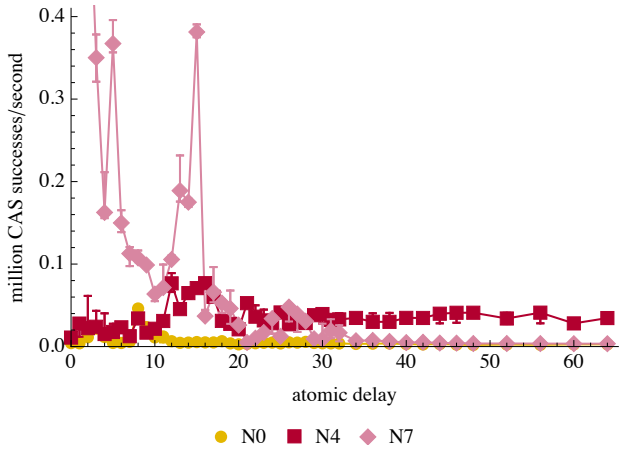


Figure 3.11: AMD throughput of CAS operations for varying atomic delay with all nodes participating. Target allocated on Node 0. Showing total throughput of Node 0 (distance 0 from target), Node 4 (distance 1), and Node 7 (distance 2).

Intel Fairness. The fairness test results on the Intel machine are shown in Figure 3.12. Only Node 0 and Node 3 are shown, as the other nodes' curves were almost exactly the same as Node 3. As could be expected, both Node 0 and Node 3 drop in throughput as the atomic delay grows, and eventually both reach approximately the same throughput.

[5]This part is truncated in the plot, to make other trends more visible.
[6]This data is not shown in the plot.

We can see that in general, fairness here is not as skewed as on AMD; at high throughputs (corresponding to low atomic delay), Node 3 outperforms Node 0 by a factor of 1.4–1.8. Both node's performance degrades quickly, though at somewhat different speeds. At an atomic delay of 34 iterations (around 75 ns), unfairness is at its worst, with Node 3 outperforming Node 0 by a factor of 12.5. However, soon after that, starting at an atomic delay of 52 iterations, the two nodes are consistently within 10% of each other in terms of their throughput.

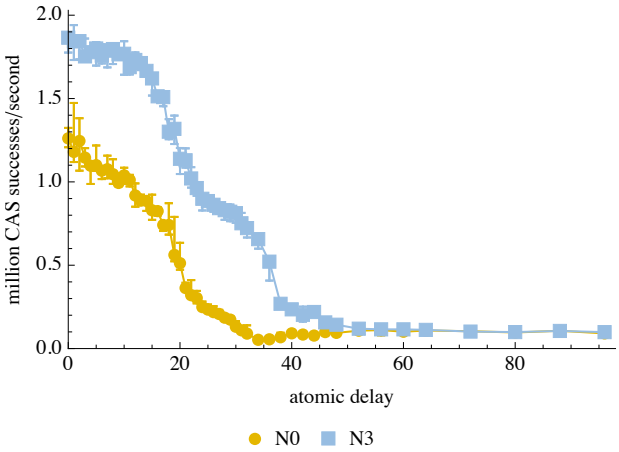


Figure 3.12: Intel throughput of CAS operations for varying atomic delay with all nodes participating. Target allocated on Node 0. Showing total throughput of Node 0 (distance 0 from target) and Node 3 (distance 1).

Fairness Takeaways. We conclude that the fairness of schedules of a lock-free algorithm is highly dependent on the algorithm itself, in particular, on the length of its atomic modify section. This observation is perhaps counter-intuitive, especially for theoreticians in the field; most literature on lock-free algorithms never accounts for 'local' work. However, the exact length of local operations within the atomic modify section can have a drastic effect on both fairness and performance. This is despite the fact that local work operates on the L1 cache and thus experience much lower latencies than memory instructions that access new or contended data. We thus recommend making efforts to minimize work in the atomic modify section when designing and implementing lock-free algorithms.

Furthermore, we note that despite fairness arbitration efforts within each node, fairness is not generally achieved among nodes. This is a similar observation to that made by Song et al. [269]. However, while they study workloads in which there is an uneven number of requests from competing nodes, we show unfairness even when all nodes issue the same number of requests. In general, to achieve better fairness even with relatively small atomic modify sections, it can be beneficial to design architectures to explicitly favor requests from the local node over those from remote nodes.

3.5.2 Focus

Recall the original intuition (Section 3.1) for why focus may be useful in a hardware schedule. Ideally, to avoid wasted work, a thread should be able to keep a cache line in its private cache for long enough to execute both the read and the CAS instructions of its *atomic modify* section in a lock-free algorithm. However, this means that depending on the length of the atomic modify section of a given algorithm, the cache line must remain in one core's cache longer for sufficient focus.

Recall that when inferring the scheduling patterns of each machine in Section 5.2, we con-

sidered the *visit length* of a cache line at each core. That is, we measured how many memory instructions a single core can execute before the cache line leaves its private cache. Note that a schedule with better focus corresponds to a schedule with longer core visits. Thus, more focus is required from the schedule the longer the atomic delay is. We say that a hardware schedule has *meaningful focus* for a given lock-free algorithm if the entire atomic modify section of the algorithm fits in a single core visit.

We now test how longer core visits observed in Section 5.2 translate to meaningful focus for lock-free algorithms. Unlike previous experiments in this chapter, we test focus using experiments with *multiple targets*. Specifically, we allocate one target on each node, and each core is assigned one target to access for the duration of the experiment. This means that each core is only directly contending with other cores accessing the same target. However, there may be indirect contention caused by traffic on the node interconnect. To exhibit a variety of core visit lengths, we run different types of experiments for AMD and Intel.

AMD Focus. Recall from Section 3.4.1 that nodes that are 2-hops away from the memory they access have longer core visits on average. To test how these longer visits translate to meaningful focus, we conduct competition experiments with three different settings. In each setting, all cores access a target that is a fixed distance away. The results of this test are shown in Figure 3.13.

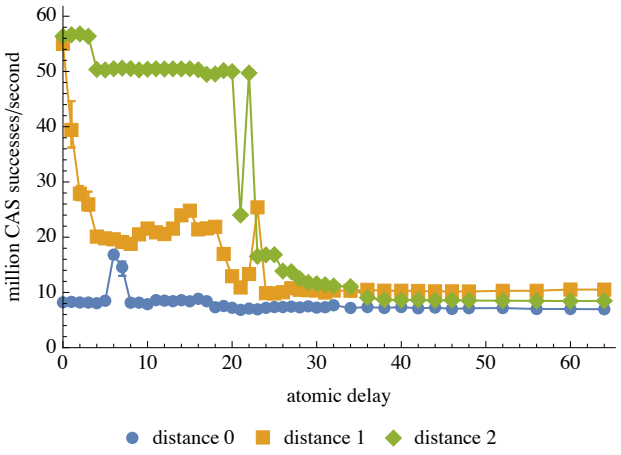


Figure 3.13: AMD throughput of CAS operations for varying atomic delay with all nodes participating. A target is allocated each node. All cores accessing a given target are in the same node. In each run, all cores access targets the same distance away.

For small atomic delays, we observe a significant difference between the three settings. In particular, both distance 1 and distance 2 placements exhibit higher throughput than distance 0. Throughput at distance 1 drops near atomic delay 18. This indicates that at this point, a thread can no longer fit both its read and its CAS into the same visit. A similar drop happens for the distance 2 placement near atomic delay 23. In contrast, it appears that the distance 0 placement never fits a read and CAS into the same visit, even with atomic delay 0.

These findings make sense in light of the results of Section 3.4.1. Specifically, as shown in Table 3.2, cores at distance 2 have longer visit lengths than those at distance 1. From the table initially appears as if distance 0 cores have visit lengths comparable to distance 1 cores. However, as shown in Figure 3.5, cores at distance 0 are frequently skipped in what is otherwise a mostly round-robin visit sequence. If we view these skips as "length 0" visits, then cores at distance 1 have visits longer than those at distance 0, whose visits can be so short that not even a single atomic instruction finishes executing.

All three thread placements eventually reach a steady throughput of around 7.2–10.3 million

successful CAS operations per second. This happens at an atomic delay of 36 iterations, roughly corresponding to 125 ns on the AMD machine (see Section 3.3). Distance 1 nodes display the highest throughput of the three categories in the steady state, outperforming distance 2 nodes by 20% and the Node 0 by 43%. This is consistent with the results from the fairness tests, but the difference in performance is smaller here.

There are some other phenomena that we do not yet know how to explain, such as the drops in throughput for distances 1 and 2 as atomic delay increases from 0 to 5 and the occasional throughput spikes. It is possible that some of these effects would be smoothed over by an experiment in which atomic delay was random rather than deterministic.

Intel Focus. Recall from Section 3.4.2 that longer visits occur on the first core visited in a node, when the cache line travels between nodes. In particular, these long core visits happen only when cores of multiple nodes are active, rather than just one node. To test the effect of longer core visits on meaningful focus in the Intel machine, we therefore compare two types of competition experiments: the first is simply using all threads of one node, and the second uses the same number of threads, but splits them across two nodes. Just like we did for AMD, we run the experiments in parallel to create interconnect traffic. The results of this test are shown in Figure 3.14.

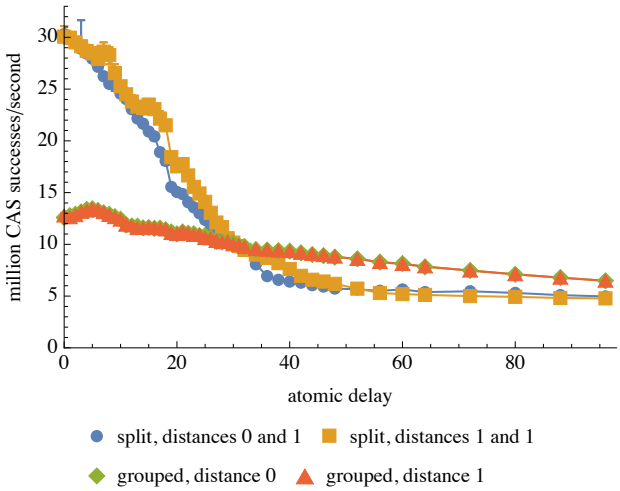


Figure 3.14: Intel throughput of CAS operations for varying atomic delay with all nodes participating. A target is allocated each node. Showing results for four different target assignments. In *grouped* assignments, all cores accessing a given target are in the same node. In *split* assignments, the set of cores accessing a given target is split evenly across two nodes.

As expected given our knowledge of Intel's schedule, it is clear that for a small atomic delay, splitting the threads across two nodes produces significantly higher throughput than having them all on one node. At around an atomic delay of 30 iterations (approximately 66 ns), the runs on a single node start outperforming the split runs. This can be attributed to the lowered contention caused by such a high atomic delay. When contention is low, the dominating factor for performance becomes the latency of accessing the memory (or the L3 cache, in this case), which is known to be much lower for local accesses than for remote accesses.

Focus Takeaways. On both machines, we observed that for experiments with low atomic delay, higher throughput occurs on schedules that we know exhibit better focus. The higher focus seems to be *meaningful* only for an atomic delay of up to approximately 25-30 iterations, indicating that algorithms with atomic modify sections of around this length or shorter can benefit from such schedules.

However, more generally, it is clear that focus in the hardware schedule is extremely helpful for throughput; it would be desirable to achieve meaningful focus even for algorithms with a longer atomic modify section. This observation was made by Haider et al. [141]. Using simulation results, they showed that it can be very beneficial to allow each thread to *lease* a cache line for a bounded amount of time, and *release* it either when that time is up, or when it finishes its atomic modify section. Our results support those of Haider et al., but on real architectures rather than simulations. That is, even when all features of an architecture interact with each other, it can be beneficial to extend the implicit lease of a cache line that memory instruction schedules provide a thread.

3.6 Related Work

Alistarh *et al.* [16] ran tests similar to our sequence experiments to verify the validity of their uniform random scheduler assumption. They ran the experiments on a single Fujitsu PRIMERGY RX600 S6 server with four Intel Xeon E7-4870 (Westmere EX) processors, but they used only one of its nodes. Our results for this setting are consistent with theirs; scheduling seems mostly uniformly random on a single Intel node. Our experiments, however, consider a much greater scope, noting when this random scheduling pattern falters.

NUMA architectures have been extensively studied. Previous works have designed benchmarks to understand the latencies and bandwidth associated with accesses to different levels of the cache and local versus remote memory on NUMA machines [138, 236, 264]. However, these papers did not consider the effect of contended workloads on NUMA access patterns.

A thorough study of synchronization primitives was conducted by David *et al.* [98]. Some of their tests are similar to ours. However, their setup is different; in all contention experiments, David *et al.* inject a large delay between consecutive operations of one thread. While we use a similar pattern for our focus and fairness experiments, we also test configurations that do not inject such delays. Thus, our work uncovers some performance phenomena that were not found by David *et al.*

Song *et al.* [269] show that NUMA architectures can have highly unfair throughput among the nodes. They also show that this unfairness does not always favor nodes that access local memory, displaying this behavior in VMs. However, they do not study lock-free algorithms or contention.

Performance prediction has been the goal of significant previous work, not only in the lock-free algorithms community [36, 71, 134, 182, 285]. Techniques range from simulation, to hand built models, to regression based models, to profiling tools. Goodman *et al.* present one such profiling tool [134]. While this produces accurate results, sometimes it is impractical to have the algorithm ready to use for profiling before performance predictions are made, since performance predictions can help develop the algorithm. Our work aims to obtain a high level performance model to guide algorithm design in its earlier stages. Furthermore, our benchmark can be used on any machine to gain an understanding of its underlying model.

3.7 Chapter Discussion

Analytical performance prediction of lock-free algorithms is a hard problem. One must consider the likely operation scheduling patterns on the machines on which the algorithm is run. Previous approaches assumed a random scheduler instead of an adversarial one, but did not show whether such an assumption is reflective of real machines.

In this chapter, we present a thorough study of scheduling patterns produced on two NUMA architectures, using Severus, our benchmarking tool [49, 50]. Our experiments uncover several phenomena that can greatly affect the schedules of lock-free algorithms and make models based solely on uniform randomness inaccurate. In particular, we show that thread placement with respect to a contended memory location can be crucial, and that surprisingly, remote threads often perform better under contention than local threads.

On both tested machines, the reason for this rise in throughput seems to stem from improved *focus*, or the increased length of visits of the cache line for cores on remote nodes. This phenomenon has been largely overlooked in literature that aims to approximate the operation scheduler, other than a few exceptions [141]. Additionally, these focus benefits come at the cost of *fairness* on modern machines; not all cores on a machine experience these beneficial longer visits.

We believe that there are several takeaways and further directions from the work presented in this chapter. Firstly, fairness is not a given. This knowledge can affect algorithm design, as well as programming frameworks chosen; in a system with low fairness, a work-stealing scheduler may be crucial for ensuring a fair allocation of parallel tasks that leads to high throughput. Secondly, this work casts doubt on previous works that assume requests for a cache line are simply handled in a random order, and shows that more careful modeling may be necessary. Furthermore, we've shown in our experiments that the length of the atomic delay (the delay between the read and the following CAS in a read-modify-CAS loop) has a significant—yet a priori unpredictable—effect on performance, since different platforms can behave drastically differently. Finally, we provide a tool that allows a user to test their platform and understand what assumptions are reasonable for them, and what factors might have the greatest effect on their algorithm's performance.

Chapter 4

Contention-Bounded Series-Parallel DAGs

4.1 Introduction

This part of the thesis has so far focused on getting a better understanding of the causes and effects of contention on modern architectures and on modeling these effects more accurately. Studying contention and refining our theoretical model is important, since most work in the literature does not theoretically analyze the performance of concurrent algorithms, instead relying exclusively on empirical evaluation.

As discussed in previous chapters, the difficulty of analyzing the performance of concurrent algorithms stems for their inherent asynchrony, which is modeled by an adversarial scheduler. This adversary can create executions in which each process executes in isolation, or have all processes contend on the same location at once. This control that the adversary possesses leads to pessimistic and inaccurate analyses. In fact, even though virtually no upper bounds are known, Fich, Hendler and Shavit [116] show that, when accounting for contention, a number of concurrent data structures, including counters, stacks, and queues, require $\Omega(n)$ time in a system with n processes. This lower bound gives a pessimistic outlook on the design of efficient data structures.

In this chapter, based on [5], we take a different approach to designing and analyzing concurrent algorithms that are provably efficient (even when accounting for contention). In particular, instead of considering a fully general concurrent setting and trying to understand likely behaviors of the scheduler (as we did in the previous two chapters), we study concurrency in a restricted but commonly used concurrent setting. We show that by considering the setting in which a concurrent algorithm is run, it is possible to design it in a way that is provably efficient. We show this by presenting an implementation of an *indicator* data structure designed to be used in nested parallel programs.

Nested parallelism is a programming paradigm in which concurrency is created under the control of programming primitives. Nested parallelism is broadly available in a number of modern programming languages and language extensions, such as OpenMP, Cilk [124], Fork/Join Java [199], Habanero Java [159], TPL [203], TBB [161], X10 [79], parallel ML [120], and parallel Haskell [181]. In these systems, nested-parallel programs can be expressed by using a primitives such as `fork−join` and `async−finish`. These primitives govern the number of processes that may participate in the execution; `fork` and `async` create new parallel tasks, allowing new

processes to join the computation to execute these tasks. Similarly, the `join` and `finish` primitives represent synchronization points, where parallel tasks are merged, terminating the processes that executed them. This allows the computation to have a tight bound on the potential amount of contention at any given point, allowing a data structure to 'prepare' for highly contended scenarios in advance. Thus, nested parallel computations yield a more structured concurrency setting, improving our ability to analyze concurrent algorithms that run in this setting.

Nested parallel computations can be modeled by a *series-parallel directed acyclic graph*, or *sp-dag*, where the vertices represent parallel tasks, and edges represent dependencies between them. A task cannot be executed unless all tasks that it depends on have completed, that is, all vertices with edges into a vertex v must have completed their execution before v can begin. The fork-join primitives allow each vertex in this dag to have at most an indegree of 2, but async-finish allows for arbitrary in-degree. Variants of dag data structures are used broadly in the implementation of modern parallel programming systems [79, 120, 124, 159, 161, 181, 199, 203].

To execute such nested parallel programs efficiently, one must determine when a vertex in the dag becomes *ready*, i.e., all of its dependencies have been executed. Such readiness detection requires a concurrent dependency counter, which we call *in-counter*, that counts the number of incoming dependencies for each task. When a dependency between a task u and v is created, the in-counter of v is incremented; when u terminates, v's in-counter is decremented; when its in-counter reaches zero,vertex v becomes ready and can be executed. In this chapter, we design an in-counter for sp-dags that is guaranteed to be efficient under all possible nested parallel executions.

Our starting point for the design of the in-counter is prior work on relaxed concurrent counters, also called *indicators*, that indicate whether a counter is positive or not. While the lower bound of Fich et al. applies to exact counters, it does not apply to indicators, leaving room for implementations that guarantee less contention. In prior work, Ellen et al. [111] present the *Scalable Non-Zero Indicator (SNZI)* data structure with the goal of achieving less contention. They prove it to be linearizable, and evaluate it empirically, but provide any analytical upper bounds. The idea of the SNZI data structure is to have a tree of *SNZI nodes*, each of which can be incremented or decremented. The change is propagated up the tree to its parent only if the *sign* of the node has changed (i.e., the node's sum changed from positive to zero or vice versa). In this way, few changes get propagated all the way up to the root, thereby preventing high contention on a single node. However, depending on the execution and where increments and decrements are initiated on the tree, the SNZI data structure could still experience significant contention. In this chapter, we present an extension to the SNZI data structure to allow it to grow dynamically at run time in response to increasing degree of concurrency.

We show that in a nested-parallel computation, in which we have a good handle on the amount of concurrency that can occur due to the fork-join and async-finish primitives, the dynamic SNZI can be used to implement in-counters with $O(1)$ amortized contention per operation.

Finally, we present an implementation and perform an experimental evaluation in comparison to prior work (Section 4.5). Our results offer empirical evidence for the practicality of our proposed techniques, showing that our approach can perform well in practice.

Our contributions are as follows.

- We present an extension to the original SNZI data structure that allows it to grow varying numbers of processes.
- We show how to use this version of SNZI to implement a non-blocking data structure for series-parallel dags that can be used to represent nested-parallel computations.
- We prove that our series-parallel dag data structure guarantees low contention under a nested parallel programming model.
- We evaluate our algorithm empirically showing that it is practical and can perform well in practice.

4.2 Background and Peliminaries

We adopt the definition of Fich *et al.* [116] for contention; the contention of an instruction i by process p in low-level execution E is the number of non-trivial[1] instructions on the same memory location as i that occur between p's most recent instruction and i in E. Recall that in this chapter, we do not restrict the adversarial scheduler beyond what is already imposed by the nested-parallel program, and therefore we do not use the definition of contention presented in Chapter 2. It would be interesting to see what bounds can be achieved for algorithms run in the nested-parallel paradigm when analyzing them under the model of Chapter 2, but this is future work beyond the scope of this thesis.

4.2.1 SP-DAGs

A nested-parallel program can be represented as a series-parallel dag, or an sp-dag, of threads, where vertices are pieces of computation, and edges represent dependencies between them. In fact, to execute such a program, modern programming languages construct an sp-dag and schedule it on parallel hardware. We present a modified implementation of an sp-dag, which incorporates a new indicator data structure for keeping track of unsatisfied dependencies. Our sp-dag data structure is provably efficient sp-dag data structure and can be used to execute nested-parallel programs. After defining sp-dags, we present in Section 4.3 our data structure for sp-dags by assuming an *in-counter* data structure that enables keeping track of the dependencies of a dag vertex. We then present our data structure for in-counters in Section 4.3.1.

Definition of SP-Dags. A *serial-parallel dag*, or an *sp-dag* for short, is a directed-acyclic graph (dag) that has a unique *source* vertex, which has indegree 0, and a unique *terminal* vertex, which has out-degree 0 and that can be defined iteratively. In the base case, an sp-dag consists of a source vertex u, a terminal vertex v, and the edge (u, v). In the iterative case, an sp-dag $G = (V, E)$ with source s and terminal t is defined by serial or parallel composition of two disjoint sp-dags G_1 and G_2, where $G_1 = (V_1, E_1)$ with source s_1 and terminal t_1 and $G_2 = (V_2, E_2)$ with source s_2 and terminal t_2, and $V_1 \cap V_2 = \emptyset$:

- **serial composition:** $s = s_1, t = t_2, V = V_1 \cup V_2, E = E_1 \cup E_2 \cup (t_1, s_2)$.

[1] recall that non-trivial instructions are those that *can* have an effect on memory, including writes, FAA, and CASes, regardless of whether or not they succeed.

```
1  type snzi_node =
2  struct {
3    c: ℕ∪{½}; initially 0 //counter
4    v: ℕ; initially 0 //version number
5    children: array of 2 snzi nodes; initially [null, null]
6    parent: snzi_node
7  }

9  void  snzi_arrive (snzi_node a) { ... }
10 void  snzi_depart (snzi_node a) { ... }
11 bool  snzi_query (snzi_node a) { ... }
```

Figure 4.1: Partial pseudocode for the SNZI data structure; full description can be found in original SNZI paper [111].

- **parallel composition:** $s, t \notin (V_1 \cup V_2)$, $V = V_1 \cup V_2 \cup \{s, t\}$, and $E = E_1 \cup E_2 \cup \{(s, s_1), (s, s_2), (t_1, t), (t_2, t)\}$.

For any vertex v in an sp-dag, there is a path from s to t that passes through v. We define the *finish vertex* of v as the vertex u which is the closest proper descendant of v such that every path from v to the terminal t passes through u.

4.2.2 The SNZI Data Structure

Concurrent counters are an important data structure used in many algorithms but can be challenging to implement efficiently when accounting for contention, as shown by the linear (in the degree of concurrency) lower bound of Fich et al [116]. Observing that the full strength of a counter is unnecessary in many applications, Ellen et al proposed *non-zero indicators* as relaxed counters that allow querying not the exact value of the counter but its sign or *non-zero status* [111]. Their non-zero indicator data structure, called SNZI, achieves low-contention by using a tree of *SNZI nodes*, each of which can be used to increment or decrement the counter. Determining the non-zero status of the counter only requires accessing the root of the tree. The tree is updated by increment and decrement operations. These operations are filtered on the way up to the root so that few updates reach the root.

Figure 4.1 illustrates the interface for the SNZI data structure and pseudo-code for the SNZI node `struct snzi_node`, the basic building block of the data structure from [111]. A SNZI node contains a counter c, a version number v, an array of children (two in our example), and a parent. The value at the counter indicates the *surplus* of arrivals with respect to the departures. The `arrive` and `depart` operations increment and decrement the value of the counter at the specified SNZI node a respectively, and `query`, which must be called at the root of the tree, indicates whether the number of `arrive` operations in the whole tree is greater than number of `depart` operations since creation of the tree.

To ensure efficiency and low contention, the SNZI implementation carefully controls the propagation of an arrival or departure at a node up the tree. The basic idea is to propagate a change to a node's parent only if the surplus at that node flips from zero to positive or vice versa. More specifically, an `arrive` at node a is called on a's parent if and only if a had surplus 0 at the

beginning of the operation. Similarly, a `depart` at a node a is recursively called on a's parent if and only if a's surplus became 0 due to this `depart`. We say that node a *has surplus due to its child* if there was an `arrive` operation on a that started in a's child's subtree.

The correctness and linearizability of the SNZI data structure as proven relies on two important invariants [111]: (1) a node has surplus due to its child if and only if its child has surplus, and (2) surplus due to a child is never negative. Together, these properties guarantee that the root has surplus if and only if there have been more `arrive` than `depart` operations on the tree.

4.3 The Series-Parallel Dag Data Structure

Figure 4.2 shows our data structure for sp-dags. We represent an sp-dag as a graph G consisting of a set of vertices V and a set of edges E. The data structure assigns to each vertex of the dag an in-counter data structure, which counts the number of (unsatisfied) dependencies of the vertex. As the code for a dag vertex executes, it may dynamically insert and delete dependencies by using several *handles*. These handles can be thought of, in the abstract, as in-counters, but they are implemented as pointers to specific parts of an in-counter data structure.

More precisely, a `vertex` consists of

- a query handle (`query`), an increment handle (`inc`), and a decrement handle (`dec`),
- a flag `first_dec` indicating whether the first or the second decrement handle should be used,
- a flag `dead` indicating whether vertex has been executed,
- the finish vertex `fin` of the vertex, and
- a `body`, which is a piece of code that will be run when the scheduler executes the vertex.

The sp-dag uses an in-counter data structure `Incounter`, whose implementation is described in the Section 4.3.1. The in-counter data structure supplies the following operations:

- `make`: takes an integer n, creates an in-counter with n as the initial count and returns a handle to the in-counter;
- `increment`: takes a vertex v, increments its in-counter, and returns two decrement and two increment handles;
- `decrement`: takes a vertex v and decrements its in-counter;
- `is_zero`: takes a vertex v and returns true iff that in-counter of v is zero.

Inspired by recent work on formulating a calculus for expressing a broad range of parallel computations [4], our dag data structure provides several operations to construct and schedule dags dynamically at run time. The operation `new_vertex` creates a vertex. It takes as arguments a finish vertex u, an increment handle i, a decrement handle d, and a number n indicating the number of dependencies it starts with. It allocates a fresh vertex, sets its `dead` and `first_dec` flags to `false`, and sets the `body` for the vertex to be a dummy placeholder. It then creates an in-counter and a handle to it. The operation returns the new vertex along with the handle to its in-counter. The handle becomes the query handle. The operation `make` creates an sp-dag consisting of a root u and its finish vertex z and returns the root u. As can be seen in the function `make`,

the increment and decrement handles of a vertex v always belong to the in-counter of v's finish vertex.

A vertex u and its finish vertex z can be thought as a computation in which u executes the code specified by its `body` and then returns to z. This constraint implies that z serially depends on u because z can only be executed after u. As it executes, a vertex u can use the functions `chain` and `spawn` to "nest" another sequential or parallel computation (respectively) within the current computation.

The `chain` operation, which corresponds to serial composition of sp-dags, nests a sequential computation within the current computation. When a vertex u calls `chain`, the call creates two vertices v and w such that w serially depends on v. Furthermore, v serially depends on u, and z serially depends on w. After calling `chain`, u terminates and thus dies. The initial in-counters of v and w are set to 0 and 1 respectively (to indicate that v is ready for execution, but w is waiting on one other vertex). The edge (u, v) is inserted to the dag to indicate a satisfied dependency between u and v.

The `spawn` operation, which corresponds to parallel composition of sp-dags, nests a parallel computation with the current computation. When a vertex u calls `spawn`, the call creates two vertices, v and w, which depend serially on u, but are themselves parallel. The operation increments the in-counter of u's finish node, to indicate a serial dependency between u's finish vertex and the new vertices v and w. Even though two new vertices are created, the in-counter is incremented once, because one of the vertices can be thought of as a continuation of u. The `increment` operation returns two new increment handles i and j and a decrement-handle pair d. The `spawn` operation then creates the two vertices v and w using the two increment handles, one for each, and the decrement-handle pair d as a shared argument. Assigning each of v and w their own separate increment handles enables controlling contention at the in-counter of u's finish node. As described in Section 4.3.1, sharing of the decrement handles by the new vertices (v, w) is critical to the efficiency. As with the `chain` operation, `spawn` must be the last operation performed by u. It therefore completes by creating the edges from u to v and w, and marking u dead.

The operation `signal` indicates the completion of its argument vertex u by decrementing the in-counter for its finish vertex v, and inserting the edge (u, v).

In terms of efficiency, apart from calls to `Incounter`, the operations of sp-dags involve several simple, constant time, operations. The asymptotic complexity of the sp-dags is thus determined by the complexity of the in-counter data structure.

4.3.1 Incounters

We present the in-counter: a time and space efficient low-contention data structure for keeping track of pending dependencies in sp-dags. When a new dependency for vertex v is created in the dag, its in-counter is incremented. When a dependency vertex in the dag terminates, v's in-counter is decremented.

Our goal is to ensure that the increment and decrement operations are quick; that is, that they access few memory locations and encounter little contention. These goals could seem contradictory at first—if operations access few memory locations then they would conflict often. Our

```
module Incounter = ... (* As defined in Figure 4.4 *)
class Dag {
  G = (V,E)
  type handle = Incounter.handle
  struct {
    handle query, inc, dec[2];
    boolean first_dec, dead;
    vertex fin;
    function body;
  } vertex;

  (vertex, handle) new_vertex (vertex u, handle i, handle d, int n) {
    V = V ∪ {v}, v ∉ V
    (v.firstDec, v.dead) = (false, false)
    v.body = { *code* } //piece of code for this vertex to run
    v.query = Incounter.make(n)
    (v.fin, v.inc, v.dec) = (u, i, d)
    return (v, v.query) }

  (vertex,vertex) make () {
    V = {v}
    E = ∅
    (v.firstDec, v.dead) = (false, false)
    v.body = { *code* } //piece of code for this vertex to run
    v.query = Incounter.make(1)
    u = new_vertex (u, v.query, v.query, 0)
    return (u,v)  }

  (vertex, vertex) chain (vertex u) {
    (w, h) = new_vertex (u.fin, u.inc, u.dec, 1)
    (v, h) = new_vertex (w, h, [h, h], 0)
    E = E ∪ { (u, v) }
    u.dead = true
    return (v, w) }

  (vertex, vertex) spawn (vertex u) {
    (d, i, j) = Incounter.increment (u.fin)
    (v, _) = new_vertex(u.fin,  i, d, 0)
    (w, _) = new_vertex(u.fin, j, d, 0)
    E = E ∪ { (u, v), (u, w) }
    u.dead = true
    return (v, w) }

  void signal(vertex u){
    Incounter.decrement (u.fin)
    E = E ∪ { (u, u.fin) }   } }
```

Figure 4.2: Pseudocode for our sp-dag data structure.

```
1 (snzi_node, snzi_node) grow (snzi_node a, float p ) {
2   heads = flip(p) //flip a p-biased coin.
3   if (heads) {
4       left = new snzi_node(0)
5       right = new snzi_node(0)
6       CAS(a.children, null, [left, right]) }
7     children = a.children
8     if (children == null) { return (a, a) }
9     return children }
```

Figure 4.3: Pseudocode for the probabilistic SNZI `grow`.

in-counter data structure circumvents this issue by ensuring that each operation accesses few memory locations that are mostly *disjoint* from those of others.

The in-counter is fundamentally a dynamic version of the SNZI tree. We begin by presenting this extension to the SNZI data structure presented by Ellen *et al.* to make it able to dynamically adjust to contention.

4.3.2 Dynamic SNZI

The SNZI data structure does not specify the shape of the tree that should be used, instead allowing the application to determine its structure. For example, if the counter is being used by p threads concurrently, then we may use a perfectly balanced tree with p nodes, each of which is assigned to a thread to perform its `arrive` and `depart` operations. This approach, however, only supports a *static* SNZI tree, i.e., unchanging over time. While a static SNZI tree may be sufficient for some computations, where a small numbers of coarse-grained threads are created, it is not sufficient for nested parallel computations, where the number of fine-grained threads vary dynamically over a wide range from just a few to very large numbers, such as millions or even billions, depending on the input size. The problem is that, with a fixed-sized tree, it is impossible to ensure low contention when the number of threads increase because many threads would have to share the same SNZI node. Conversely, it is impossible to ensure efficiency when the number of threads is small, because there may be many more nodes than necessary.

To support re-sizing the tree dynamically, we extend the SNZI data structure with a simple operation called `grow`, whose code is shown in Figure 4.3. The operation takes a SNZI node as argument, determines whether to extend the node (if the node has no children), and returns the (possibly newly created) children of the node. If the node already has children, pointers to them are returned, but they are left unchanged. If the node does not have children, then they are locally allocated, and then the process tries to atomically link them to the tree. They are initialized with surplus 0 so that their addition to the tree does not affect the surplus of their parent. The `grow` operation also takes a probability p, which dictates how likely it is to create new children for the node. The operation only attempts to create new children if the node does not already have children and it flipped heads in a coin toss with probability p for heads. The idea is that this probabilistic growing allows an application to control the rate at which a SNZI tree grows without changing the protocol for some of the threads using it. Such probabilistic growing is useful when one wants a balance between low contention and frequent memory allocation.

The right probability threshold to use, however, may depend not only on the application, but also on the system.

If node a does not have any children at the end of a `grow` operation, we have the operation return two pointers to a itself. This return value is convenient for the in-counter application that we present in the rest of the chapter. Other applications using dynamic SNZI may return anything else in that case.

The `grow` operation may be called at any time on any SNZI node. To keep the flexible structure of the SNZI data structure, we do not specify when to use the `grow` operation on the tree, instead leaving it to the application to specify how to do so. For example, in Section 4.3.1, we show how to use the `grow` operation to implement a low-contention dependency counter for dags. It is easy to see that the operation does not break the linearizability of the SNZI structure; it is completely independent from the *count*, *version number*, and *parent* fields of nodes already in the tree, which are the only fields that affect the correctness and linearizability.

We note a key property of the `grow` operation. We would like to ensure that, given a probability p, when `grow` is called on a childless node, only $1/p$ such calls will return no children in expectation, regardless of the timing of the calls. That is, even if all calls to `grow` are concurrent, we want only $1/p$ of them to return no children. This property is ensured by having the coin flip happen *before* reading the value of the $children$ array to be returned. This guarantees that any adversary that does not know the result of local coin flips cannot cause more than $1/p$ calls to return no children in expectation.

Dynamically shrinking the SNZI tree is more difficult, because we need to be sure that no operation could access a deallocated node. In general, one may easily and safely delete a SNZI node from the tree if the following two conditions hold: (1) its surplus is 0, and (2) no thread other than the one deleting the node can reach it. The second condition is much trickier to ensure. In Section 4.4 we show how to deallocate nodes in our dependency counter data structure.

4.3.3 Choosing Handles to the In-Counter

To ensure disjointness of memory accesses, our in-counter data structure assigns different *handles* to different dag vertices. These handles are pointers to SNZI nodes within the in-counter, dictating where in the tree an operation should begin. Thus, if two processes have different handles, their operations will access different memory locations.

The in-counter data structure ensures the invariant that only the leaves have a surplus of zero. This allow us to exploit an important property of SNZI: operations complete when they visit a node with positive surplus, effectively stopping propagation of the change up the tree. Specifically, an increment that starts at a leaf completes quickly when it visits the parent. To maintain this invariant, we have to be careful about which SNZI nodes are decremented.

Figure 4.4 shows the pseudo-code for the in-counter data structure.

The in-counter's interface is similar to the original SNZI data structure: the operation `make` creates an instance of the data structure and returns a handle to it. The operation `increment` is meant to be used when a dag vertex spawns. It takes in a dag vertex, and uses its increment handle to increment the SNZI node's surplus. Before doing so, it calls the `grow` operation. Intuitively, this growth request notifies the tree of possible contention in the future and gives it a chance to grow to accommodate a higher load. The `increment` operation returns handles to other nodes

```
1  class Incounter{
2    type handle = snzi_node //A handle is a snzi node

4    float p = *prob*  //Growth probability: architecture specific constant.

6    //Make a new counter with surplus n.
7    handle make(n) { snzi_make(n)  }

9    //Auxiliary function: select decrement handle.
10   handle claim_dec(vertex u) {
11     if (CAS(u.firstDec, false, true)) { u.dec[0]    }
12     else u.dec[1]    }

14   //Increment u's target dependency counter.
15   (handle[2], handle, handle) increment (vertex u) {
16     (a,b) = grow(u.inc, p)
17     (i1,i2) = (a,b)
18     if (u is a left child) {d2 = a }
19     else  {d2 = b }
20     snzi_arrive (d2)
21     d1 = claim_dec(u)
22     return [[d1,d2],i1,i2] }

24   //Decrement u's counter.
25   void decrement (vertex u) {
26     d = claim_dec(u)
27     snzi_depart(d)}

29   //Check that u's counter is zero.
30   boolean is_zero(u) { return snzi_isZero(u.query)}
31 }
```

Figure 4.4: Pseudocode for the in-counter data structure.

in the SNZI tree, that is, two decrement handles and two increment handles, indicating where the newly spawned children of the dag vertex should perform their increment or decrement. The operation `decrement` is meant to be used when a dag vertex signals the end of its computation. It takes in dag vertex, and uses its decrement handle to decrement the surplus of a previously incremented node.

Since the `increment` operation is the only thing that can cause growth in the SNZI tree, the structure of the tree is determined by the operations performed on the in-counter structure. The tree is therefore not necessarily balanced. However, as we establish in Section 4.4, the operations' running time does not depend on the depth of the tree. In fact, we show that the operations complete in constant amortized time and also lead to constant amortized contention.

To understand the implementation, it is helpful to consider the way the sp-dag structure uses the in-counter operations. The `increment` operation is called by a dag vertex when the vertex calls the `spawn` operation, which uses the vertex's increment handle to determine the node in the SNZI tree, where an arrive operation will start. To find the place, the `increment` operation first uses a `grow` operation to check whether the SNZI node pointed at by the handle has children. Additionally, the operation creates new children if necessary (line 16). Intuitively, we create new SNZI nodes to reduce contention by using more space. Thus, if the increment handle has children, we should make use of them and start our `arrive` operation there. This is exactly what the `increment` operation does (line 20). Note that, in case the `grow` operation does not return new children, we simply have the `arrive` operation start at the increment handle. The `increment` operation returns two increment handles and two decrement handles; one of each for each of the dag vertex's new children.

The in-counter algorithm makes use of an important SNZI property: decrements at the top of the tree do not affect any node below them. Furthermore, if there is a depart operation at SNZI node a, it cannot cause a to phase change as long there is surplus anywhere in a's subtree. These observations lead to the following intuition: to minimize phase changes in the SNZI tree, priority should be given to decrementing nodes closer to the root.

Thus, each dag vertex stores two decrement handles rather than just one. These two decrement handles are shared with the vertex's sibling, and they are ordered: the first handle always points to a SNZI node that is higher in the tree than the second. Decrement handle is needed by the `decrement` operation (line 26) and the `increment` operation. Recall that an `increment` always returns two decrement handles. One of these decrement handles always points to the SNZI node on which the `increment` operation started its `arrive`. However, the other one is inherited from the parent dag vertex (the one that invoked the `increment`). To preserve the invariant, the incrementing vertex needs to claim a decrement handle (line 21). The two decrement handles returned are always ordered as follows: first, the one inherited from the parent, and, then, the one pointing at the freshly incremented SNZI node. In this way, we guarantee that the first decrement handle in any pair points higher in the tree than its counterpart. When a decrement handle is needed, the two dag vertices determine which handle to use through a test-and-set. The first of the two to use a decrement handle will take the first handle, thus ensuring that higher SNZI nodes are decremented earlier. We rely on this property to prove our bounds in Lemma 4.4.8.

Note that, in an `increment` operation, the decrement handle is claimed only *after* the `arrive` has completed. This key invariant helps to ensure that phase changes rarely occur in the SNZI tree, thus leading to fast operations.

4.4 Correctness and Analysis

We first prove that our accusations data structure is linearizable and then establish its time and space efficiency.

We say that the accusations is correct if any `is_zero` operation at the root of the tree correctly indicates whether there is a surplus of `increment` operations over `decrement` operations. To prove correctness, we establish a symmetry with the original SNZI data structure. Note that each `increment`, `decrement`, and `is_zero` operation calls the corresponding SNZI operation (i.e., `arrive`, `depart`, `query` respectively) exactly once. We define the linearization points of each operation as that of the corresponding SNZI operation.

Since the correctness condition for the accusations and for SNZI are the same, we have the following observation.

Observation 4.4.1. *An execution of the accusations is linearizable if the corresponding SNZI execution is linearizable.*

At this point, we recall that any SNZI run is linearizable if there are never more departs than arrives on any node, i.e., surplus at any node is never negative [111]. We show that this invariant indeed holds for any valid execution of the accusations data structure, and therefore that any valid execution is linearizable. We define a valid execution as follows.

Definition 4.4.2. *An accusations execution is* valid *if any handle passed as argument to the* `decrement` *operation was returned by a prior* `increment` *operation. Furthermore, that handle is passed to at most one* `decrement` *operation.*

Lemma 4.4.3. *Any valid accusations execution is linearizable.*

Proof. Observe that every `increment` operation generates one new decrement handle, and that handle points to the node at which this `increment`'s arrive operation was called. Thus, in valid executions, any `depart` on the underlying SNZI data structure (which is always called by the `decrement` operation) has a corresponding `arrive` at an earlier point in time. Therefore, there are never more departs than arrives at any SNZI node, and the corresponding SNZI execution is linearizable. From Observation 4.4.1, the accusations execution is linearizable as well. □

It is easy to see that any execution of the accusations that only accesses the object as prescribed by the sp-dag algorithm is valid. From this observation, the main theorem follows immediately.

Theorem 4.4.4. *Any execution of the accusations through sp-dag accesses is linearizable.*

4.4.1 Running Time

To prove that the accusations is efficient, we analyze shared-memory steps and contention separately: we first show that every operation takes an amortized constant number of shared memory steps, and then we show that each shared memory location experiences constant amortized contention at any time. In our analysis, we do not consider `is_zero`, since it does not do any non-trivial shared memory steps.

For the analysis, consider an execution on an sp-dag, starting with a dag, consisting of a single root vertex and its corresponding finish vertex. Further, observe that this finish vertex

has a single SNZI node as the root of its accusations. Note that all shared memory steps in the execution are on the accusations. We begin by making an important observation.

Lemma 4.4.5. *There exist at most one increment and one decrement handle pointing to any given SNZI node.*

To prove this lemma, note that every `increment` operation creates new children for the SNZI node whose handle is used, and thus does not produce another handle to that node. A simple induction, starting with the fact that the root node only has one handle pointing to it, yields the result.

We now want to show that every operation on the accusations performs an amortized constant number of shared memory steps. First, note that the only calls in the accusations operations that might result in more than a constant number of steps are their corresponding SNZI operations. The SNZI operations do a constant number of shared memory steps per node they reach, but they can be recursively called up an arbitrarily long path in the tree, as long as nodes in that path phase change due to the operation. That is, frequent phase changing in the SNZI nodes will cause operations on the data structure to be slower. In fact, we show that on average, the length of the path traversed by an `arrive` or `depart` operation is constant, when those SNZI operations are called only through the accusations operations.

We say that a dag vertex is *live* if it is not marked dead and a handle is live if it is owned by a live vertex. We have the following lemma about live vertices.

Lemma 4.4.6. *If dag vertex v is live, then it has not used `claim_dec` to claim a decrement handle.*

Keeping track of live vertices in the dag is important in the analysis, since by Lemma 4.4.6, these are the vertices that can use their handles and potentially cause more contention.

To show that operations on SNZI nodes are fast, we begin by considering executions in which only increment operations are allowed, and no decrements happen on the accusations. Our goal is to show that the accusations's SNZI tree is relatively saturated, so that `arrive` operations that are called within a `increment` only access a constant number of nodes. After we establish that increment operations are fast when decrements are not allowed, we bring back the decrement operations and see that they cannot slow down the increments.

Lemma 4.4.7. *Without any decrement operations, when there are no `increment`s mid-execution, only leaves of the in-counter's SNZI tree can have surplus 0.*

Proof. The proof is by induction on the number of increment operations called on the accusations. The tree is initialized with one node with surplus 1. Thus, when there have been 0 increments, no node has surplus 0. Assume that the lemma holds for accusationss that have had up to k increments on them. Consider an accusations that has had k increments, and consider a new `increment` operation invoked on node a in the tree. If a is not a leaf, the `grow` operation does not create new children for a, and the tree does not grow. Thus, the lemma still holds. If a is a leaf, the `increment` operation creates two children for a and starts an arrive operation on one of the children. As a consequence of the SNZI invariant, we know that the surplus of a increased by 1. After the `increment`, regardless of what its surplus was before this operation, a cannot have surplus 0, and the lemma still holds. □

By Lemma 4.4.7, we know that, ignoring any effect that decrements may have, the tree has

surplus in all non-leaf nodes. Recall that an `arrive` operation that reaches a node with positive surplus will terminate at that node. Thus, any `arrive` operation invoked on this tree will only climb up a path until it reaches the first node that was not a leaf before this `increment` started. That is, since each `increment` only expands the tree by at most one level, any `arrive` operation will reach at most 3 nodes.

We now bring back the `decrement` operations. The danger with decrements is that they could potentially cause non-leaf nodes to phase change back to 0 surplus, meaning that an `arrive` operation in that node's subtree might have to traverse a long path up the SNZI tree. Our main lemma shows that traversal of a long path can never happen; if a decrement causes a vertex's surplus to go back to 0, then no subsequent increment operation will start in that node's subtree.

Lemma 4.4.8. *If an in-counter node a at any point had a surplus and then phase changed to 0 surplus, then no live vertices in the dag point anywhere in its subtree.*

Proof. We prove this by induction on the size of a's subtree. Assume that node a in the in-counter has a surplus at time t, and consider the decrement operation that caused a's surplus to become 0.

If a has no children, we only need to consider the handles that point directly at a, since it has no subtree. Note that it must be the case that a was incremented by an `arrive` operation that started there, in a `increment` operation that had an increment handle to a's parent. During that operation, a decrement handle to a was created, and placed as the *second* of the pair of decrement handles returned. If a was now decremented, *both* dag vertices that shared these decrement handles must have claimed them, and thus, by Lemma 4.4.6, neither of them is now live. In particular, this means that the dag vertex that owned the increment handle to a is dead. So, no live dag vertex has a handle to a.

Assume that the lemma holds for nodes with subtrees of size at most k. Now consider a SNZI node a which has a subtree of size $k + 1$. Consider the decrement operation that caused a's surplus to change to 0. There are two possible cases: either (1) this decrement's first depart is at a, or (2) it started at one of a's descendants.

CASE 1. This decrement started at a. Note that a has children, but neither of them has a surplus at the time this decrement operation starts, since otherwise, a would have a surplus due to its children, and would not phase change. Note that for a to have children, an `increment` operation must have started at a and used `touch` to create a's children. This same `increment` then `arrive`d at one of a's children. So at least one of a's children must have had surplus at some point, and now neither of them do. By the induction hypothesis, if both of them had surplus at some point, then no live dag handles point anywhere in a's childrens' subtrees now. Furthermore, since by Lemma 4.4.5 only one increment handle and one decrement handle are ever created for a, no handle in the dag points anywhere in a's subtree (including a itself).

If only one of a's children ever had surplus, by the induction hypothesis, that child now has no handles pointing to it or its subtree. Furthermore, for that child's surplus to become 0, both of the dag vertices that were created in the `spawn` operation that created a's children must have used their decrement handles. Note that this includes the only dag vertex that had an increment handle to a's other child. Thus, neither of a's children have any live handles pointing to them, and since a's handles have been used, no live handles now point to anywhere in a's subtree.

CASE 2. The decrement that caused a's phase change started in a's subtree, but not at a itself.

Let a_r denote a's right child and a_l denote its left child. Without loss of generality, assume that the decrement started in a_r's subtree.

First note that a_r must have had surplus before this decrement started, since no node can have a negative surplus. We also know that after this decrement, a_r has a surplus of 0, since it must have phase changed in order for a `depart` to reach a. Thus, by the induction hypothesis, no live handles point to a_r.

We now only have to consider a_l to see whether there are still live handles pointing to that subtree. We know that a_l does not have surplus, since otherwise a would not phase change due to this decrement. Assume by contradiction that there is a live handle pointing to a_l. Then by the induction hypothesis, a_l has never had a surplus, and there is a live increment handle to it. Note that, by the algorithm, the dag vertex that has the increment handle to a_l must have a decrement handle pointing to a_r. By Lemma 4.4.6, this decrement pointer has never been used, thus contradicting our earlier conclusion that a_r has 0 surplus. □

Note that Lemma 4.4.7 and Lemma 4.4.8 immediately give us the following important property.

Corollary 4.4.9. *No `increment` operation can invoke more than 3 arrive operations on the SNZI tree.*

Proof. We already saw that by Lemma 4.4.7, if no decrements happen, then each `increment` operation can invoke at most 3 `arrive` operations.

Now consider decrements as well. Note that if a non-leaf node has surplus 0 and there is no `increment` mid-operation at that node, then it must have previously had surplus (again by Lemma 4.4.7). By Lemma 4.4.8, no new `increment` operations can happen on that node's subtree. Thus, even allowing for decrement operations, no `increment` operation can invoke more than 3 arrives on the SNZI tree. □

To finish the proof, we amortize the number of `depart`s invoked by `decrement`s against the `arrive`s invoked by `increment`s. Recall that the number of `depart`s invoked on a SNZI tree is at most the number of `arrive`s invoked on it. We get the following theorem:

Theorem 4.4.10. *Any operation performed on the accusations itself calls an amortized $O(1)$ operations on the SNZI tree.*

We now move on to analyzing the amount of contention a process experiences when executing an operation on the accusations. Note that there have been several models suggested to account for contention, as discussed in Section 4.6. In our contention analysis, we say that an operation contends with a shared memory step if that step is non-trivial, and it could, in any execution, be concurrent with the operation at the same memory location. Our results are valid in any contention model in which trivial operations do not affect contention (i.e. such as stalls [116], the CRQW model [132], etc.).

Theorem 4.4.11. *Any operation executed on the accusations experiences $O(1)$ amortized contention.*

Proof. We show something stronger: the maximum number of operations that access a single SNZI node over the course of an entire dag computation (other than `is_zero` operations on the root) is constant.

Consider a node a in the SNZI tree. By Lemma 4.4.5, only one `increment` operation ever starts its `arrive` at a. That `increment` has exactly one corresponding `decrement` whose `depart` starts at a. By the SNZI algorithm, an `arrive` at a's child only propagates up to a if that child's surplus was 0 before the `arrive`. By Lemma 4.4.8, this situation can only happen once; if a's child's surplus returns to 0 after being higher, the counter will never be incremented again. Thus, each of a's children can only ever be responsible for at most two operations that access a; the initial `arrive` and the final `depart` from that subtree.

In total, we get a maximum of 6 operations that access a over the course of the computation. Note also that the first `arrive` at a node always strictly precedes any other operation in that node's subtree. Thus, concurrent `arrive` operations are not an issue. Therefore, any operation on the in-counter can be concurrent with at most 4 other operations per SNZI node it accesses. By Theorem 4.4.10, every operation on the accusations accesses amortized $O(1)$ nodes. Therefore, any such operation experiences amortized $O(1)$ contention. □

4.4.2 Space Bounds

Shrinking the tree by removing unnecessary nodes is tricky because, in its most general form, an ideal solution likely requires knowing the future of the computation. Knowing where in the tree to shrink requires knowing which nodes are not likey to be incremented in the future. For the specific case of the `grow` probability of 1, we establish a safety property that makes it possible to keep the data structure compact by removing SNZI nodes that correspond to deleted sub-graphs of the sp-dag.

We would like to show that our in-counter data structure does not take much more space than the dag computation uses on its own. To do so, we will show when it is safe to delete SNZI nodes in relation to the state of the dag vertices that have handles to them. First, however, we note that even without the dynamic shrinking of the SNZI tree, there are never more nodes in the in-counter than the total number of dag vertices created in an execution. This is because whenever a dag vertex spawns to create two more children, the SNZI tree also grows by two nodes.

However, we can do better than that by deleting SNZI nodes that will never be used again. For this, we begin with the following lemma:

Lemma 4.4.12. *Any node whose surplus was positive and then returned to 0 may be deleted safely from the SNZI tree.*

This lemma is a direct consequence of Lemma 4.4.8. Any node whose surplus goes back to 0 has no live handles pointing to it, and is thus safe for deletion.

Using this knowledge, we can shrink the SNZI tree as the computation progresses. To be able to express the next theorem, we need to define when a vertex u in the dag has *finished*. This is a recursive definition:

- **Base Case.** If u has no children, then it is *finished* when it signals the end of its computation.
- **Iterative Case.** If u has children, then is is *finished* when both of its children have finished.

We note the following important lemma which relates finished vertices to the state of the execution in an sp-dag that uses in-counters.

Lemma 4.4.13. *If a vertex u has finished, then the decrement handle that it claimed has been used.*

Proof. We prove this by induction on the number of descendants u has. If u doesn't have any descendants, then we know it had to use its claimed decrement handle before it finished, since by definition, it finishes when it signals, and to signal, it needs to call the `decrement` operation on the in-counter.

Assume the lemma holds for any dag vertex u with up to k descendants. Let u be a dag vertex with $k+1$ descendants. By the induction hypothesis, both of its children have used their claimed decrement handles. Note that of those two handles, one was claimed by u and passed down to its children. Thus, u's decrement handle has been used. □

Now, we can relate finished vertices to deallocation.

Theorem 4.4.14. *Consider a dag vertex u and its increment handle, pointing to SNZI node a. If u has finished, then any node in a's subtree, excluding a itself, can be safely deleted from the SNZI tree.*

Proof. We prove this by induction on the size of a's subtree. If a doesn't have children, then we are done - no nodes are deleted when u finishes.

Assume that if u is finished, and has an increment handle to SNZI node a, which has a subtree of size at most k, then a's entire subtree, excluding a itself, can be deleted. Let u be a finished vertex whose increment handle points to a node a with a subtree of size $k+1$.

Note that by definition, if u is finished, then both of its children are finished as well. Note that, by the algorithm, u's children's increment handles must point to a's children in the SNZI tree. Thus, by the induction hypothesis, a's entire subtree, excluding itself and its children, can be deleted safely. In particular, it means that neither of a's children have surplus due to their children. Note that only one of a's two children had an `arrive` operation start on it (this was done by u's increment operation, which created a's children). Thus, if neither of them has surplus due to their children, at most one of them has surplus. Note that the decrement handle to that child was owned by one of u's children. By Lemma 4.4.13, since both of u's children have finished, they have both used their decrement handles. Thus, a's child has been decremented, and now has surplus 0. Furthermore, even if a's other child never had surplus, the dag vertex that owned its increment handle has used its decrement handle, so a's other child can never grow. Thus, both of a's children can be safely deleted. □

4.5 Experimental Evaluation

We report an empirical comparison between our in-counter algorithm, the simple fetch-and-add counter, and an algorithm using fixed-size SNZI trees. Our implementation consists of a small, C++ a library that uses a state-of-the-art, implementation of a work-stealing scheduler [3]. Overall, our results show that the simple, fetch-and-add counter performs well only when there is one core, the fixed-size SNZI trees scale better, but our in-counter algorithm outperforms the others when the core count is two or more.

Implementation. Our implementation of core SNZI operations `snzi_arrive` and `snzi_depart` follows closely the algorithm presented in the original SNZI paper, except for two differences. First, we do not need `snzi_query` because readiness detection is performed via the return result of `snzi_depart`. This method suffices because only the caller of `snzi_depart` can bring the count to zero. Second, our `snzi_depart` returns `true` if the call brought the counter to zero. To control the grain of contention, we use the probabilistic technique presented in Section 4.3.2, with probability $p := \frac{1}{25c}$, where c is the number of cores. The idea of using c is to try to keep contention the same as the number of cores increase. We chose the constant factor 25, because it yields good results, but as our Threshold study, described below, shows many other constants also yield good results, e.g., any constant in the range $2.5c \leq p \leq 25c$ yields qualitatively the same results. The implementation of the sp-dags and in-counter data structures (Section 4.3) build on the SNZI data structure. The sp-dags interface enables writing nested-parallel programs, using various constructs, such as fork-join and async-finish; we use the latter in our experiments.

We compare our in-counter with an atomic, fetch-and-add counter because the fetch-and-add counter is optimal for very small numbers of cores. For higher number of cores, we designed a different, SNZI-based algorithm that uses a fixed-depth SNZI tree. This algorithm gives us another point of comparison, by offering a data structure that uses the existing state of the art more directly. The fixed-depth SNZI algorithm allocates for each finish block a SNZI tree of $2^{d+1} - 1$ nodes, for a given depth d. To maintain the critical SNZI invariant that the surplus of a SNZI node never becomes negative, the fixed-depth SNZI algorithm ensures that every `snzi_depart` call targets the same SNZI node that was targeted by a matching `snzi_arrive` call. To determine which SNZI node to be targeted by a `snzi_arrive` call, we map DAG vertices to SNZI nodes using a hash function to ensure that operations are spread evenly across the SNZI tree.

Experimental Setup. We compiled the code using GCC `-O2 -march=native` (version 5.2). Our machine has four Intel E7-4870 chips and 1Tb of RAM and runs Ubuntu Linux kernel v3.13.0-66-generic. Each chip has ten cores (40 total) and shares a 30Mb L3 cache. Each core runs at 2.4Ghz and has 256Kb of L2 cache and 32Kb of L1 cache. The machine has a non-uniform memory architecture (NUMA), whereby RAM is partitioned into four banks. Pages are, by default in our Linux distribution, allocated to banks according to the first-touch policy, which assigns a freshly allocated page to the bank of the processor that first accesses the page. An alternative policy assigns pages to banks in a round-robin fashion. We determined that, for the purposes of our experiments, the choice of NUMA policy has no significant impact on our

```
fun fanin_rec(n)
  if n ≥ 2
    async fanin_rec(n/2)
    async fanin_rec(n/2)

fun fanin(n)
  finish {fanin_rec(n)}
```

Figure 4.5: Fan-in benchmark.

```
fun indegree2(n)
  if n ≥ 2
    finish {
      async indegree2(n/2)
      async indegree2(n/2)
    }
```

Figure 4.6: Indegree-2 benchmark.

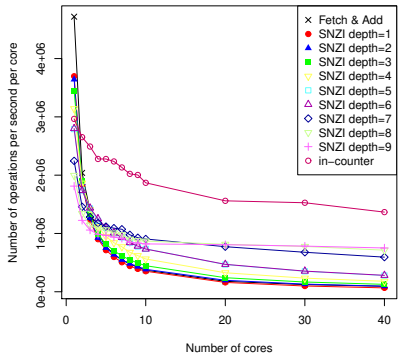


Figure 4.7: Fanin benchmark with $n = 8$ million, & varying number of processors. Higher is better.

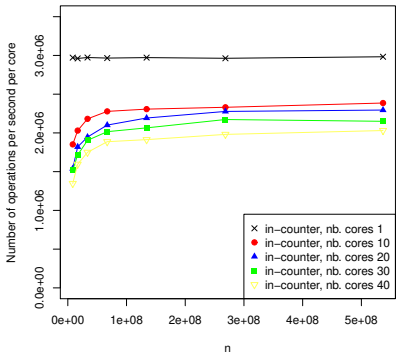


Figure 4.8: Fanin benchmark, varying the number of operations n. Higher is better.

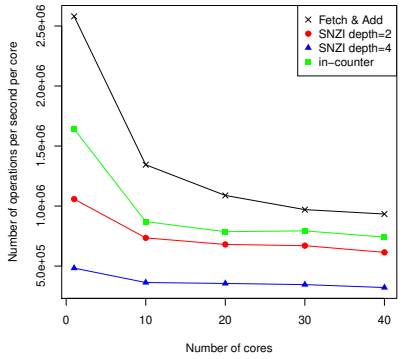


Figure 4.9: Indegree-2 benchmark, using number of operations $n = 8$ million. Higher is better.

main results. The experiment we performed to back this claim is described in more detail in the appendix. For all other experiments, we used the round-robin policy.

Benchmarks:`fanin` and `indegree2`. To study scalability, we use the two small benchmarks shown in Figures 4.5 and 4.6. The `fanin` benchmark performs n async calls, all of which synchronize at a single finish block, making the finish block a potential source of contention. The `fanin` benchmark implements a common pattern, such as a parallel-for, where a number of independent computations are forked to execute in parallel and synchronize at termination. Our second benchmark, `indegree2` implements the same pattern as in `fanin` but by using binary fork join. This program yields a computation dag in which each finish vertex has indegree 2.

Scalability Study. Figure 4.7 shows the scalability of the `fanin` benchmark using different counter data structures. With one core, the Fetch & Add counter performs best because there is no contention, but with more, Fetch & Add gives worst performance for all core counts (this pattern is occluded by the other curves). For more than one core, our in-counter performs the best. The fixed-depth SNZI algorithm scales poorly when depth is small, doing best at the tree depth of 8, where there are enough nodes to deal with contention among 40 cores. Increasing the depth further does not improve the performance, however.

Size-Invariance Study. Theorem 4.4.11 shows that the amortized contention of our in-counter is constant. We test this result empirically by measuring the running time of the `fanin` benchmark for different input parameters n. As shown in Figure 4.8, for all input sizes considered and for all core counts, the throughput is within a factor 2 of the single-core (no-contention) FAA counter. The throughput suffers a bit for the smaller values of n because, at these values, there is not enough useful parallelism to feed the available cores.

Low-Indegree Study. Our next experiment examines the overhead imposed by the in-counter by using the `indegree2` benchmark shown in Figure 4.6. The results, illustrated in Figure 4.9, show that our in-counter data structure is within a factor 2 of the best performer, the fetch-and-add counter. For SNZI, we only considered small-depths, since larger ones took too long to run. With the fixed-depth SNZI, the benchmark creates many finish blocks and thus many SNZI trees, becoming inefficient for large trees.

Threshold Study. In Section 4.3.2, we presented a technique to control the grain of contention by expanding the SNZI tree dynamically. Here, we report, using the `fanin` benchmark, the results of varying the range of probabilities $p = \frac{1}{threshold}$ for different settings of $threshold$. As shown in Figure 4.10, essentially any threshold between 50 and 1000 works well. We separately checked that on our test machine using a constant threshold such as 100 or 1000 yields good results when using any number of cores. On a machine, perhaps with many more cores, the best setting might be different or might also depend on the number of cores used.

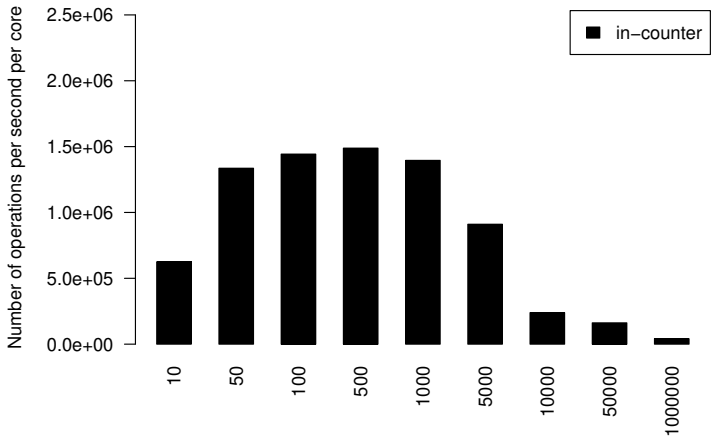


Figure 4.10: Threshold experiment. Each bar represents a different setting for $p = \frac{1}{threshold}$. All runs were performed using 40 cores. Higher is better.

4.6 Related Work

Upper bounds for several non-blocking data structures, accounting also for contention, have been proven for several tree- and list-based search structures [114, 121, 244]. In these upper bounds, contention factors in as a linear additive term, which is consistent with established lower bounds for many problems in the general concurrency model [115, 116, 151, 168]. In contrast, for relaxed counters, we show that contention for series-parallel programs can be upper bounded by a constant.

Complexity models for analyzing contention and techniques for designing algorithms for reducing contention has also been important subject of study [110, 130, 132, 177, 217] Contention has also been identified as an important practical performance issue in multiprocessor systems. Techniques that reduce contention by detecting and introducing wait periods or using tokens to be passed between threads proved to be essential for reducing detrimental effects of contention [7, 19]. Managing contention in software-transactional memory may require contention managers that schedule transactions carefully to minimize aborted transactions [143, 152, 260, 266].

In this chapter, we consider non-blocking, or lock-free algorithms. Since our upper bounds are quite good, $O(1)$, our algorithms guarantee that each operation completes quickly in our relaxed concurrency model. Wait-free data structures can guarantee similar properties in the general concurrency model [148, 149]. Such data structures were traditionally considered to be impractical, though recent results show that they may be also be practical [186, 273].

Part II

Non Volatile Random Access Memory

Chapter 5

Delay-Free Simulations on NVRAM

5.1 Introduction

In the previous part of the thesis, we studied shared memory, and adjusted the classic model to reflect parts of shared memory behavior that are not usually taken into account. Notably, we did not consider new memory technology, but instead deepend our understanding of existing hardware. In contrast, in this part, which consists of only one chapter, we study a completely new technology, called Non-Volatile Random Access Memory (NVRAM), that changes the way in which shared memory can be used. This chapter is based on the work presented in [47].

NVRAM offers byte-addressable persistent memory at speeds comparable with DRAM. This memory technology now can co-exist with DRAM on the newest Intel machines, and may largely replace DRAM in the future. Upon a system crash in a machine with NVRAM, data stored in main memory will not be lost. However, without further technological advancements, caches and registers are expected to remain volatile, losing their contents upon a crash. Thus, NVRAM yields a new model and opportunity for programs running on such machines—how can we take advantage of persistent main memory to recover a program after a fault, despite losing values in cache and registers?

Programs for persistent storage in the past have prioritized design concerns that are not relevant for use on NVRAM; since disk accesses are in block-granularity, and are significantly more expensive than DRAM accesses, persistent programs were primarily designed to minimize block accesses. However, NVRAM accesses are not nearly as expensive, being similar in latency to DRAM. Futhermore, the byte-addressability of NVRAM means that previous algorithms that assumed block-granularity of accesses to persistent memory are not only optimizing for the wrong metric, but also potentially wrong if run on NVRAM. Whereas before, blocks could be updated on persistent storage atomically, on NVRAM we may experience a fault in the middle of such an update. However, this shift in design concerns is worth the effort, since it offers the potential to have persistent programs that are orders of magnitude faster than those that rely on disk. A lot of work has thus focused on developing persistent algorithms for index trees [81, 201, 204, 245, 276], lock-based data structures [74, 237], and lock-free data structures [89, 100, 122] for this new NVRAM setting.

A natural question that arises is whether we can find general mechanisms that would port

memory-only (non-persistent) algorithms for current machines over to the new persistent setting. One approach has been the development of persistent transactional memory frameworks [92, 188, 215, 228]. This can be an effective approach, although it does not handle code between transactions. Another approach to achieve arbitrary persistent data structures is the design of persistent universal constructions [89, 166]. In particular, Cohen *et al.* [89] present a universal construction that only requires one flush per operation, thereby achieving optimality in terms of flushes. However, universal constructions often suffer from poor performance, because they sequentialize accesses to the data structure, and lead to executions with high contention. Furthermore, universal constructions are only applicable to data structures with clearly defined operations, and cannot apply to a program as a whole. For these reasons, even a seemingly efficient universal construction leaves more to be desired.

Another downside of these approaches is that they don't make use of existing efficient concurrent algorithms designed for DRAM (without persistence). As discussed in the previous part of the thesis, the performance of concurrent algorithms heavily depends on factors like contention, disjoint access parallelism, and remote accesses. While these factors are difficult to theoretically characterize, they are monumental in their effect on an algorithm's performance. Decades of work on the design of concurrent algorithms has yielded carefully engineered solutions that take these factors into account. When creating algorithms for the persistent setting, it is hence important to be able to preserve the structure of these tried and tested efficient concurrent algorithms in their persistent counterparts.

In this chapter, we therefore take a different appraoch; we consider simulators that take any concurrent program and transform it by replacing each of its instructions with a simulation that has the same effect. The simulation provides a mapping between the original version and the new version of the algorithm, preserving its structure. We assume the Parallel Persistent Memory (PPM) model [30, 59]. The model consists of n processes, each with a fast local ephemeral memory of limited size, and sharing a large persistent memory. The model allows for each process to fault and restart (independently or together). On faulting all the process's state and local ephemeral memory are lost, but the persistent memory remains. On restart, each process has a location in persistent memory that points to a context from which it loads its registers, including program counter, and restarts.

In this model, we define *persistent simulations*, which exhibit a tradeoff between their *computation delay*, meaning the overhead introduced by the simulation in a run without faults, and their *recovery delay*, which is the maximum time the simulation takes to recover from a fault. We also consider the *contention delay*, which characterizes the extra contention that may be introduced by the simulation. Our first result is the presentation of a general persistent simulator, called the *Constant-Delay Simulator*, that takes any concurrent program using Reads and compare-and-swap (CAS) operations, and simulates it with constant computation and recovery delays.

Theorem 5.1.1. *Any concurrent program that uses Reads and CAS operations can be simulated in the persistent memory model with constant computation delay and constant recovery delay.*

The Constant-Delay Simulator is achieved by transforming code to be *idempotent*, meaning that it can be repeated several times and only take effect once. To make code idempotent, we use two techniques: code encapsulation and recoverable primitives.

The idea behind code encapsulation, introduced by Blelloch *et al.* [59], is that the program is broken up into contiguous chunks of code, called *capsules*. Between every pair of capsules, information about the state of the execution is persisted. When a crash occurs in the middle of a capsule, recovery reloads information to continue the execution from the beginning of the capsule. This means that some instructions may be repeated several times, but only within a single capsule. Thus, this reduces the problem of making the entire code idempotent to the problem of ensuring that each of the smaller capsules is idempotent. Blelloch *et al.* [59] gave sufficient conditions to ensure that sequential code is idempotent. However, repetitions of concurrent code can be even more hazardous, as other processes may observe changes that should not have happened. We therefore formalize what it means for a capsule to be correct in a concurrent setting, and show how to build correct concurrent capsules.

To build correct capsules for general concurrent programs, we must be able to determine whether a modification of a shared variable can safely be repeated. This can be problematic, because often, the execution of a process depends on the return value of its accesses to shared memory. A bad situation can occur if a process has already made a persistent change in shared memory, but crashed before it could persist its operation's return value [59]. Attiya *et al.* [30] consider this problem and define *nesting-safe recoverable linearizability (NRL)*, a correctness condition for persistent objects that allows them to be safely nested. In particular, Attiya *et al.* [30] introduced the *recoverable CAS*, a primitive which ensures that if a compare-and-swap by process p has successfully changed its target, this fact will be made known to p even if a crash occurs.

At a high-level we show how to combine the ideas of capsules with recoverable primitives in a careful way to achieve our results for programs with shared reads and CASes. We use a modification of this recoverable CAS primitive in our capsules to ensure that a program can know whether it should repeat a CAS. We show that the recoverable CAS algorithm satisfies a stronger property than NRL, allowing the recovery to be called even if a fault occurs after the operation has terminated. This property is very important for use in capsules, because after a fault, all operations of the capsule must be recovered, rather than just the most recent one.

In this way, we achieve our persistent simulations by encapsulating code and replacing atomic instructions with their recoverable counterparts. This leads to a tradeoff between computation and recovery delay when it comes to capsule size; each capsule introduces some overhead in normal execution when persisting the next checkpoint, but decreases the recovery delay, since the we only have to repeat at most one capsule when recovering from a fault. In a nutshell, our first result, the Constant-Delay Simulator, shows that we can have constant computation overhead per capsule, and that we can have constant-sized capsules, leading to both constant computation and recovery delay. In practice, however, faults are relatively rare. It is thus important to minimize the computation delay introduced by a persistent simulator, even at the cost of increased recovery time. After we present the Constant-Delay Simulator, we therefore present optimizations that can be applied to the simulator to decrease the computation delay. The first such optimization is just as general as the Constant-Delay Simulator in that it applies to any concurrent program. The difference is that we use fewer capsules; we show where boundaries between capsules can be removed, creating capsules that are larger (not necessarily constant sized), to arrive at a smaller computation delay but a larger recovery delay. The second optimization applies to a large class of lock-free data structures called *normalized data structures* [274]. In this setting, we show how

to further reduce the number of capsules and thus the computation delay.

We test our simulations by applying them to the lock-free queue of Michael and Scott (MSQ) [232], and comparing their performance with two other state-of-the-art implementations: one using the transactional memory framework Romulus [92], and the other a hand-tuned detectable queue, known as the LogQueue [122]. We did not expect general constructions to match the performance of specialized implementations, but it turns out to be very close. The LogQueue outperforms our transformations on 2-6 running threads, but only by an average of 5%; our most optimized transformation even outperforms the LogQueue on 1, and 7-8 threads. In comparison, the original MSQ is usually between 3.57x to 1.9x faster than the LogQueue, showing that the inevitable cost of persistence outweighs the extra cost paid for generality by our transformations. We further show that our optimizied transformation outperform Romulus for the thread counts we tried.

In summary, in this chapter we present the following contributions.

- We define persistent simulations, which consider computation and recovery delay, thereby providing a measure of how faithful a simulation is to the original program, and how fast it can recover from crashes.
- We present a constant-computation and constant-recovery delay simulation that applies to any concurrent program.
- We show optimized simulations that trade-off computation delay for recovery delay, both for general programs and for normalized data structures.
- We show that our transformations are practical by comparing experimentally to state-of-the-art persistent algorithms.

5.2 Model and Preliminaries

We use the Parallel Persistent Memory (PPM) model [30, 59]. It consists of a system of n asynchronous processes $p_1 \dots p_n$. Each process may access a *persistent* shared memory of size M with Read and CAS instructions, as well as a smaller private *ephemeral* memory, which can be accessed with standard RAM instructions. The return value of instructions on persistent memory are stored in ephemeral variables. A process can *persist* the contents of its ephemeral memory by writing them into a persistent memory location. The ephemeral memory is explicitly managed; it does not behave like a cache, in that no automatic evictions occur. Memory operations are sequentially consistent, and all memory locations can hold $O(\log M)$ bits.

Each process may *fault* between any two instructions. Upon a fault, the contents of a process's private ephemeral memory is lost, but the persistent memory remains unchanged. After a process fault, it *restarts*. To allow for a consistent restart, each process has a fixed memory location in the persistent memory referred to as its *restart pointer location*. This location points to a context from which to restart (i.e., a program counter and some constant number of register values). On restart this is reloaded into the registers, much like a context switch. Processors can checkpoint by updating the restart pointer. Furthermore a process can know whether it has just faulted by calling a special `fault()` function that returns a boolean flag, and resets once it is called. We call programs that are run on a processor of such a machine *persistent programs*.

For our simulations we also consider a standard concurrent RAM (C-RAM) consisting of n asynchronous processes with local registers and a shared memory of size M. As with the PPM we assume the C-RAM supports Reads and CAS instructions on the shared memory, that the memory is sequentially consistent, and registers and memory locations can hold $O(\log M)$ bits.

Recall from Section 1.5 that an *high-level execution* involves invocation and response events for the operations of each process. In this chapter, we add another type of event that executions may have; the *fault* event. Fault events are process-specific. On a fault event C_i, the process p_i loses all information stored in its ephemeral variables (but shared objects remain unaffected).

5.2.1 Capsules

Our goal is to create concurrent algorithms that can recover their execution after a fault. The main idea in achieving this is to periodically persist *checkpoints*, which record the current state of the execution, and from which we can continue our execution after a fault. We call the code between any two consecutive checkpoints a *capsule*, and the checkpoint itself a *capsule boundary*. At a boundary, we persist enough information to continue the execution from this point when we recover from a fault. This approach was use by Blelloch *et al.* in [59] and similar to approaches by others [90, 101, 218, 222]. An *encapsulation* of a program is the placement of such boundaries in its code to partition it into capsules.

Capsule Correctness. When executing recoverable code that is encapsulated, it is possible for some instructions to be repeated. This happens if the program crashes in the middle of a capsule, or even at the very end of it before persisting the new boundary, and restarts at the previous capsule boundary. To reason about the correctness of encapsulated programs after a crash, we define what it means for a capsule to be *correct* in a concurrent setting, intuitively meaning that it is *idempotent*, i.e. that can be repeated safely.

Definition 5.2.1. *An instruction I in a low-level execution E is said to be* invisible *if E remains legal even when I is removed.*

Definition 5.2.2. *A capsule C inside algorithm A is* correct *if:*

1. *Its execution does not depend on the local values of the executing thread prior to the beginning of the capsule, and*
2. *For any high-level execution E of A in which C is restarted from its beginning at any point during C's interval an arbitrary number of times, there exists a set of invisible instructions performed in E as part of C such that when they are removed, C only executes once in the corresponding low-level execution.*

Definition 5.2.3. *A program is* correctly encapsulated *if all of its capsules are correct.*

5.3 k-Delay Simulations

We are interested in efficiently simulating arbitrary computations on a reliable concurrent RAM (C-RAM) on a faulty PPM machine. Due to factors like contention and disjoint access parallelism, which are difficult to theoretically analyze, it is desirable to be able to preserve the structure of tried and tested efficient concurrent algorithms in their persistent counterparts. We

formalize this notion of 'preserving structure' with the definitions of *computation delay*, *recovery delay* and *contention delay*. Roughly, the first refers to the the number of instructions on the PPM required to simulate each instruction on the C-RAM when there is no fault, the second refers to the number of instructions needed to recover from each fault, and the third takes into account the delay caused by concurrent accesses to the same object.

Before talking about delays, we begin with the definition of a *linked simulation*, which provides a mapping from the instructions of the simulated code to those of the simulation itself, thereby preserving the original structure. We allow for arbitrary computations on the machine that is being simulated, as long as the machine is sequentially consistent.

Definition 5.3.1 (Linked Simulation)**.** *Consider a concurrent* source *machine S with n processes for which each process executes a sequence of atomic operations, and a concurrent* target *machine T, also with n processes. A* linked simulation *of S on T is a simulation that allows for any computation on S, preserves its behavior, and for each process p the execution of p on T can always be partitioned, contiguously, so that:*

1. *there is a one-to-one order-maintaining correspondence between these partitions and operations of p in some execution on S, and*
2. *each partition atomically simulates the corresponding source operation, linearized at some point between the partition's first and last operation.*

We refer to each partition on each process as a step *of the simulation.*

Note that our definition includes all local operations as well as any access to shared objects. The simulated operations could include linearizable operations on shared objects, or could just be machine instructions. Our first simulation operates at the granularity of machine instructions We say a step of a linked simulation is *fault free* if no fault happened on the step's process on T during that step. For a sequence of instructions s on a single process of either the source or target machine let $t(s)$ be the number of instructions in s, i.e., the time.

Definition 5.3.2 (Computation Delay)**.** *A linked simulation has k* computation delay *if for each fault-free step s of the target machine simulating an operation o of the source machine, $t(s) = O(t(o) + k)$.*

Ben-David et al. use a similar concept of delay for measuring the efficiency of transactions [48]. Persistent computations are tightly coupled with their recovery mechanisms. When discussing a computation for a persistent setting, it is important to also discuss how it recovers from faults. Note that, if all processes fault together during a system fault, simply running a concurrent program as is in a persistent setting yields a trivial 1 computation delay simulation of itself; all steps of the program remain exactly the same. However, upon a fault, the entire computation has to be restarted, and all progress is completely lost. Thus, the recovery time of this 'simulation' is unbounded; it grows with the length of the execution. We therefore also formalize the notion of a *recovery delay*; how long it takes for a persistent program on any process to recover from a fault (processes can fault independently).

Definition 5.3.3 (Recovery Delay)**.** *A linked simulation of a source machine S on a faulty target machine T has k* recovery delay *if each fault that occurs within a step of the simulation on T adds at most k instructions to the step.*

Note that the k for computation delay and recovery delay need not be the same. Furthermore, our simulations guarantee a constant, in general the k could be a function of other parameters of

the machine or computation (e.g. input size or number or processes).

A stronger notion of a k computation delay simulation is one in which the amount of contention experienced by an algorithm cannot grow by more than a factor of k either. Accounting for contention helps to capture the structure of an algorithm, since scalability is highly associated with keeping contention as low as possible on all accesses. To be able to discuss contention formally, we follow the definition of contention presented by Dwork *et al.* in [110]; the amount of contention experienced by an operation op on object O is the number of *responses* to operations on O received between the invocation and response of op[1]. We note that this measure of contention can also be seen as a special case of the model introduced by Ben-David and Blelloch [43] and discussed in Chapter 2 of this thesis, in which invocations correspond to an operation being made active, and conflicts are defined between all operations on the same location. We now extend the definition of computation delay to account for contention.

Definition 5.3.4 (Contention Delay)**.** *A linked simulation has* k contention delay *if the following condition holds:*

for any operation op *during the simulated computation on the source* S*, if* op*'s the contention is* C*, then the corresponding step in the simulation on the target* T *has contention at most* $k \times C$*.*

In this chapter, we consider persistent simulations that have small computation, recovery and contention delay. We say that an algorithm is X*-delay free* if it has c X delay for a constant c, where X is one of computation, contention, or recovery.

5.4 Building Blocks

5.4.1 Capsule Implementation

We now briefly discuss how to implement the capsule boundaries themselves, and in particular, how we handle persisting the ephemeral variables between capsules. We keep a copy of each ephemeral variable in persistent memory. At a high level, we update the persistent copy of each ephemeral variable with its current value at the end of each capsule, and read these values into ephemeral memory after a fault. However, we must be careful to avoid inconsistencies that could arise if a fault occurs after updating some, but not all, of the persistent copies. Such inconsistencies could occur if ephemeral variables suffer *write-after-read* conflicts in a capsule [59]. In a nutshell, these conflicts occur if a variable can be read and then written to in the same capsule. If the write gets persisted and then a fault occurs, causing the code to repeat itself from the read instruction, then the read sees a different value than it did originally. This could mean that the next repetition of the capsule may behave differently from the first, and could violate idempotence. These conflicts can be avoided if it can be guaranteed that after a new value is persisted, the program will never repeat an earlier instruction that reads it. Note that in general, it is not a problem if an ephemeral variable suffers a write-after-read conflict, since their writes are to ephemeral memory and thus are not persistent. However, since we persist their new values at the capsule boundary, we must be careful not to overwrite their previous value if we might need to use it again.

[1]For atomic memory operations, which don't have invocations and responses, we can split them into two instructions, one for the invocation and one for the response.

Thus, for ephemeral variables that are read and subsequently written to in this capsule, we do not write out their new values into their persistent copies right away. Instead, we keep two persistent *write buffers*, along with a persistent bit indicating which one of them is currently *valid*. At the boundary at the end of the capsule, the *invalid* write buffer is cleared, and for each ephemeral variable v that has a write-after-read conflict, a tuple of the form $(v, newVal)$ is written into it, where $newVal$ is the value of v at the end of the capsule. After writing out all of these tuples to the write buffer, the validity bit for the write buffers is flipped. At the beginning of each capsule, before executing any of its code, the *valid* write buffer is read, and the persistent copy of each ephemeral variable that appears in it is updated with this ephemeral variable's value in the buffer.

Note that persisting the values of ephemeral variables that are either not updated at all, or updated before being read in a given capsule, is not difficult. We don't have to do anything at the capsule boundary for an ephemeral variable that was not updated in this capsule. For ephemeral variables that were updated before being read in this capsule, we can simply copy in their new value into their persistent copy, without going through the write buffer. This is because the current capsule's execution does not depend on the value of such an ephemeral variable. This copying is done before the validity bit of the write buffer is flipped.

5.4.2 Recoverable Primitives

Recall that the return values of instructions on persistent memory are stored automatically on ephemeral variables. This represents registers in real machines, which store return values of memory accesses, but which remain volatile despite new NVRAM technology. This means that such return values may be lost upon a fault. For example, consider a CAS operation that is applied to a shared memory location. It must atomically read the location, change it if necessary, and return whether or not it succeeded. If a fault occurs immediately after a CAS is executed, the return value could be lost before the process can view it. When the process recovers from the fault, it has no way of knowing whether or not it has already executed its CAS. This is a dangerous situation; repeating a CAS that was already executed, or skipping it altogether, can render a concurrent program incorrect. In fact, any primitive that changes the memory suffers from the same problem.

This issue was pointed out by Attiya *et al.* in [30]. To address the problem, they present several recoverable primitives, among them a *recoverable CAS algorithm*. This algorithm is an implementation of a CAS object the addition that the read and CAS operaitons on it also have read.recover and CAS.recover parts to them, which must be invoked immediately after a fault, and may not be invoked at any other time. The idea of the algorithm is that when CASing in a new value, a process writes in not only the desired value, but also its own ID. Before changing the value of the object, a process must *notify* the process whose ID is written on the object of the success of its CAS operation. The recovery of the CAS operation checks this notification to see whether its last CAS has been successfully executed. Attiya *et al.* show that their recoverable CAS algorithm satisfies *nesting-safe recoverable linearizability (NRL)*, intuitively meaning that as long as recovery operations are always run immediately after faults, the high-level execution is linearizable. Attiya *et al.*'s algorithm uses classic CAS as a base object, and assumes that CAS operations are ABA-free. This is easy to ensure by using timestamps.

It turns out Attiya *et al.*'s recoverable CAS algorithm satisfies strict linearizability [8], a stronger correctness property than NRL. The main difference between the two properties is that while NRL only allows recovering operations that were pending when the fault happened, strict linearizability is more flexible. This means that we can define the recovery to work even on operations that have already completed at the time of the fault. This property is very important for use in the transformations provided in the rest of the chapter, in which we may not know exactly where in the execution we were when a fault occurred. To satisfy strict linearizability, we need to tweak the recoverable CAS algorithm, to include the use of sequence numbers on each CAS. In contrast to Attiya *et al.*, we treat the recovery as another operation of the recoverable CAS object, whose sequential specification is as follows.

Each *Recover*(i) operation R returns a sequence number seq and a flag f with the following properties:

- If $f = 1$, then seq is the sequence number of the last successful CAS operation with process id i.
- If $f = 0$, all successful CAS operations before R with process id i have sequence number less than seq.

We also further modify the recoverable CAS algorithm to create a version that has constant recovery time (instead of $O(P)$), and uses less space ($O(P)$ instead of $O(P^2)$). The modification is simple; just like Attiya *et al.* [30], we have processes CAS in their id along with their desired value, and we rely on a notification mechanism for recovering return values. However, we also include a sequence number, which is CASed in along with the process id and value. This allows us to simplify the notification mechanism; instead of the 2-D array of notification slots (one slot per pair of processes) employed by Attiya *et al.*'s algorithm, we reduce the notification slots to a single slot per process, which is where others notify that process of the result of its own CAS attempts. Process p_i writes the sequence number of its next CAS operation into its slot before executing its CAS, and others only notify it if the sequence number they observed is the same as the sequence number currently written in its slot. Algorithm 5.1 shows the pseudocode for this algorithm. Note that it implements a recoverable CAS object using $O(1)$ steps for all three operations. We use capital letter operation names to discuss the operaitons being implemented, and lower case names to describe machine instructions used on the base objects.

Correctness of our Recoverable CAS. We now prove that Algorithm 5.1 is correct by showing it satisfies *strictly linearizability* [8]; that is, that all operations are linearizable, and are either linearized before a fault event, or not at all. This condition lets us safely repeat an operation if the recovery says that it didn't happen.

We first prove the following key property about the `Notify` method.

Lemma 5.4.1. *Let N be an instance of a `Notify` method that reads $x = \langle *, seq, i\rangle$. Then after N's execution, $A[i] = \langle seq, 1\rangle$ or $A[i] = \langle seq', *\rangle$, where $seq' > seq$, and $*$ is used as a placeholder for any value.*

Proof. We first show that at the first step of N, the sequence number in $A[i]$ is at least seq. This is because for x to contain $\langle *, seq, i\rangle$, process p_i must have executed a CAS with sequence number

Algorithm 5.1: Recoverable CAS algorithm

```
1  class RCas {
2    ⟨Value, int, int⟩ x;  //shared persistent
3    ⟨int, bool⟩ A[P];     //shared persistent

5    Value Read(){
6      ⟨v, *, *⟩ = x;
7      return v;}

9    ⟨Value, int, int⟩ Notify(){
10     ⟨v, pid, seq⟩ = read(x);
11     CAS(A[pid], ⟨seq, 0⟩, ⟨seq, 1⟩);
12     return ⟨v, pid, seq⟩; }

14   bool Cas(Value a, Value b, int seq, int i){
15     ⟨v, pid, seq'⟩ = Notify();
16     if(v != a) return false;
17     A[i] = ⟨seq, 0⟩;        // announce
18     return CAS(x,⟨a, pid, seq'⟩,⟨b, i, seq⟩);}

20   ⟨int, bool⟩ Recover(int i){
21     Notify();
22     return A[i];}}
```

seq. By the code, p_i first had to update $A[i]$ to contain $< seq, 0 >$ before CASing in this value on x.

Note that $A[i]$ can only change in two ways: (1) process p_i can change $A[i]$ to a higher sequence number. In this case, the lemma holds. (2) Some process p_j can change $A[i]$ by notifying p_i. Note that any `Notify` attempt that reads sequence number $seq' \neq seq$ would fail to change $A[i]$. In particular this means that notifications cannot cause the sequence number in $A[i]$ to become smaller. So, the only way $A[i]$ can change by a through a notification (without p_i also changing it in a CAS operation) is to contain $< seq, 1 >$.

Thus, if some change occurs on $A[i]$ during N's execution, the lemma holds. Assume by contradiction that no change occurs during N's execution. In this case, N's CAS contains the correct old value, and therefore must succeed. This contradicts the assumption that no change occurs (i.e., N itself makes the change). □

Now we are ready to prove that Algorithm 5.1 is strictly linearizable. Its linearization points are also given by the following lemma.

Lemma 5.4.2. *Algorithm 5.1 is a strictly linearizable implementation of a recoverable CAS object with the following linearization points:*

- *Each CAS operation that sees $v \neq a$ on line 16 is linearized when it performs line 10. Otherwise, it is linearized when it performs line 18.*
- *Each Read operation is linearized when it performs line 6.*
- *Each Recover operation is linearized when it returns.*

Proof. We first note that all linearization points defined in the lemma statement can only be executed by the process that invoked the operation. Therefore, if that process stalls, its operation will not be linearized. This property reduces strict linearizability to proving linearizability.

To show that CAS and Read operations linearize correctly, we can ignore operations on A because they do not affect the return values of these operations. At every configuration C, the variable x stores the value written by the last successful CAS operation linearized before C. This is because the value of x can only be changed by the linearization point of a CAS operation. So x always stores the current value of the persistent CAS object.

Each Read operation R is correct because R reads x at its linearization point and returns the value that was read. Each CAS operation C is either linearized on line 10 or line 18. If C is linearized on line 10, then it behaves correctly because x does not contain the expected value at the linearization point of C. Suppose a is the value C expects and b is the value it wants to write. If C is linearized on line 18, then we know that $x = \langle a, j, s' \rangle$ at line 10 of C for some process id j and sequence number s'. If C is successful, then x contained the expected value at the linearization point of C which matches the sequential specifications. Otherwise, we know that x changed between lines 10 and 18 of C. Since this recoverable CAS object can only be used in a ABA free manner, we know that x does not store the value a at the linearization point of C, so C is correct to return false and leave the value in x unchanged.

In the remainder of this proof, we argue that the *Recover* operation is correct. Let $seq(C)$ represent the sequence number of a CAS operation C. Let R be a call to *Recover* by process p_i and let C be the last successful CAS operation by process p_i linearized before the end of R. We just need to show that R either returns $\langle seq(C), 1 \rangle$ or it returns $\langle s', 0 \rangle$ for some sequence number $s' > seq(C)$.

First note that the sequence number in $A[i]$ is always increasing, as argued in the proof of Lemma 5.4.1. Thus, the sequence number returned by R is at least $seq(C)$ since $A[i]$ is set to $\langle seq(C), 0 \rangle$ on the line before the linearization point of C. First we show that R cannot return $\langle s', 1 \rangle$ for any $s' > seq(C)$ and then we show that R cannot return $\langle seq(C), 0 \rangle$.

Now we show that R cannot return $\langle s', 1 \rangle$ for any $s' > seq(C)$. R returns the value of $A[i]$, so suppose for contradiction that $A[i] = \langle s', 1 \rangle$ at some configuration before the end of R. Then there must have been a `Notify` method that set $A[i]$ to this value. This notify operation must have seen $x = \langle *, s', i \rangle$ when it performed its first step. This means that a successful CAS by p_i with sequence number $s' > seq(C)$ has been linearized which contradicts our choice of C.

From Lemma 5.4.1, we can complete the proof by showing that there is a `Notify` method that reads $x = \langle *, seq(C), i \rangle$ and completes before R reads $A[i]$. To show this we just need to consider two cases: either x is changed between the linearization point of C and line 10 of R, or it is not. In the second case, the `Notify` call performed by R reads $x = \langle *, seq(C), i \rangle$. In the first case, some other successful CAS operation must have been linearized after the linearization point of C and before R reads $A[i]$. Let C' be the first such CAS operation. Then we know that $x = \langle *, seq(C), i \rangle$ between the linearization points of C and C'. Furthermore, in order for C' to have been successful, the read on line 10 must have occurred after the linearization point of C (otherwise C' would not see the most recent process id and sequence number). Therefore the notify operation on lines 10 and 11 of C' see that $x = \langle *, seq(C), i \rangle$ and this notify completes before R reads $A[i]$ as required. □

Lemma 5.4.2, plus the fact that each operation only performs a constant number of steps, immediately lead to the following theorem.

Theorem 5.4.3. *Algorithm 5.1 is a strictly linearizable, contention-delay-free and recovery-delay-free implementation of a recoverable CAS object.*

5.5 Persisting Concurrent Programs

One way to ensure that a program is tolerant to faults is to place a capsule boundary between every two instructions. We call these *Single-Instruction* capsules. Can this guarantee a correctly encapsulated program? Even with single-instruction capsules, maintaining the correctness of the program despite faults and restarts is not trivial. In particular, a fault could occur after an instruction has been executed, but before we had the chance to persist the new program counter at the boundary. This would cause the program to repeat this instruction upon recovery.

Trivially, if the single instruction I in a capsule C does not modify persistent memory, then I is invisible, and thus C is correct. But what if I does modify persistent memory? A private persistent write is invisible because the process simply overwrites the effect of its previous operation, and no other process could have changed it in between. So, we only have CASs left to handle. This is where we employ the recoverable CAS operation.

We replace every CAS object in the program with a recoverable CAS. We show that it is safe to repeat a recoverable CAS if we wrap it with a mechanism that only repeats it if the recovery operation indicates it has not been executed; any repeated CAS will become invisible to the higher level program. When recovering from a fault, we simply call a `checkRecovery` function, that takes in a sequence number, and calls the `Recover` operation of the recoverable CAS object. The `checkRecovery` function returns whether or not the CAS referenced by the sequence number was successful. If it was, then we do not repeat it, and instead continue on to the capsule boundary. Otherwise, the CAS is safe to repeat. Pseudocode for the `checkRecovery` function is given in Algorithm 5.2.

With this mechanism to replace CAS operations, single-instruction capsules are correct. The formal proof of correctness is implied by the proof of Theorem 5.6.1, which we show later.

Algorithm 5.2: Check Recoverable CAS

```
1  bool check_recovery(RCas X, int seq, int pid){
2    ⟨last, flag⟩ = X.Recover(pid);
3    if (last >= seq && flag == true) return true;
4    else return false;  }
```

We now show that this transformation applied to any concurrent program C is a contention-delay free simulation of C.

Theorem 5.5.1. *For any concurrent algorithm A written in the C-RAM model, if A_s is the program resulting from encapsulating A using single-instruction capsules, then A_s is a c-contention-delay, c'-recovery-delay simulation of A, where c and c' are constants.*

To prove the theorem, we first show a useful general lemma, that relates the way a simulated object is implemented to the contention-delay of a simulation algorithm.

Lemma 5.5.2. *Let A_s be a k-computation-delay simulation of A. If for every two base objects O_1 and O_2, the set of primitive objects used to implement O_1 is disjoint from the set used to implement O_2 in A_s, then A_s is a k-contention-delay simulation of A.*

Proof. Consider an execution E of A in which an operation op by process p on object O_1 experiences $k * C$ contention. Since O_1 is implemented with primitive objects that are not shared with any other object, all contention experienced by op must be from other operations that are accessing O_1. Note that each step by another process accessing O_1 can cause at most one contention point for op. Thus, there must be at least $k * C$ steps by other processes on the primitive objects op is accessing within op's interval. Since A is a k-delay simulation of A', and O_1's primitive objects are not shared with any other base object, there must be at least C other accesses of O_1 that are concurrent with op. So, E must map to an execution E' of A' in which all C of these accesses to O_1 happen before op's corresponding access, but after the last operation of p. Therefore, in E', op experiences at least C contention. □

Proof of Theorem 5.5.1. Since each recoverable CAS and each capsule can be used to recover in constant time, it is easy to see that A_s has constant recovery delay. We implement each base object O of A by calling the operations of O, followed by a capsule boundary. For CAS, we implement it by replacing the CAS object with a recoverable CAS object and also calling a capsule boundary. Because both the recoverable CAS algorithm and the capsule boundary take a constant number of instructions, we have shown that our transformation is a k-computation-delay simulation of A. Furthermore, each recoverable CAS object uses primitive objects that are unique to it, and not shared with any other object. Note that while the capsule boundary does use primitive objects that are shared among other capsule boundaries, the capsule boundaries are in fact local operations, since each process uses its own space for persisting the necessary data. So capsule boundaries do not introduce any contention. The rest of the proof therefore follows from Lemma 5.5.2. □

5.6 Optimizing the Simulation

Although capsule boundaries only consist of a constant number of uncontended instructions, they can still be expensive in practice, as they require persisting several pieces of data and use up to two fence instructions. We now discuss how to reduce the number of required capsule boundaries in a program, while still maintaining correctness. Fewer capsule boundaries means more instructions per capsule, so more progress could be lost due to a fault.

5.6.1 CAS-Read Capsules

We begin by showing that at a high level, as long as there is only one CAS operation per capsule, and this operation is the first of the capsule, the program remains correctly encapsulated.

We note that Blelloch *et al.* [59] comprehensively showed how to place capsule boundaries in non-racey persistent code to ensure idempotence. Their guideline is to create capsules that avoid *write-after-read* conflicts (See Section 5.4.1). Therefore, in addition to the capsule boundaries dictated by instructions on shared memory as outlined above, we also place a capsule boundary

between a read of a persistent private location in memory and the following write to that location. However, note that we don't always add this extra boundary; if a capsule begins with a persistent write of a private variable v, any number of reads and writes to v may be executed in the same capsule, since there is no write-after-read conflict.

Another potentially dangerous situation arises when we have a branch that depends on a shared memory read and the two paths after the branch write to different persistent memory locations. If this is all placed within a single capsule, then faulting could cause an incorrect execution where both persistent memory locations are written to. For example, there could be a fault immediately following the write to one location, and after recovering, the process could see a different shared value and decide to follow the other branch, leading to a write on the other persistent memory location.

We call this construction a *CAS-Read* capsule. We also allow for capsules that do not modify any shared variables at all. We call such capsules *Read-Only* capsules. Intuitively, all read operations are always invisible, as long as their results are not used in a persistent manner. So, a capsule that has at most one recoverable CAS operation, followed by any number of shared reads, is correct.

Note that we assume that every process has a sequence number that it keeps locally, and increments once per capsule. At the capsule boundary, the incremented value of the sequence number is persisted (along with other ephemeral values, like, for example, the arguments for the next recoverable CAS operation). Therefore, all repetitions of a capsule always use the same sequence number, but different capsules have different sequence numbers to use.

We now describe in more detail how to use the recovery function of the recoverable CAS object. This assumption is realistic, since in most real systems, there is a way for processes to know that they are now recovering from a fault. We use the `fault()` function to optimize some reads of persistent memory— if we are recovering from a fault, we read in all ephemeral values we need for this capsule from the place where the previous capsule persisted them. Otherwise, there is no need to do so, since they are still in our ephemeral memory. We show pseudocode for the CAS-Read capsule in Algorithm 5.3. Read-Only capsules are a subset of the code for CAS-Read capsules.

We now show that the CAS-Read capsule is correct. This fact trivially implies that Read-Only capsules are correct as well, so we do not prove their correctness separately. We wrap up this section by showing that a transformation that applies CAS-Read and Read-Only capsules is a c-contention-delay simulation for constant c. Intuitively, removing capsule boundaries can only improve the contention delay.

Note that the definition of correctness is with respect to an algorithm that contains the capsule. Here, we prove the claim in full generality; we show that this capsule is correct in *any* algorithm that could use it. For this, we argue that its repeated operations are invisible in any execution, despite possible concurrent operations. Note that this implies correctness for any context in which the capsule might be used.

Theorem 5.6.1. *If C is a CAS-Read capsule, then C is a correct capsule. We also require that each process increments the sequence number before calling CAS.*

Proof. Consider an execution of C in which the capsule was restarted k times due to faults.

Algorithm 5.3: CAS-Read Capsule

```
1   if (fault()){
2     *Read all vars persisted by the
3      previous capsule into ephemeral memory.*
4     seq = seq+1;
5     flag = check_recovery(X, seq, pid);
6     if (!flag){ //Operation 'seq' wasn't done
7       c = X.Cas(exp,new,seq, pid); }
8     else {
9       c = 1; }  }
10  else {
11    seq = seq+1;
12    //exp and new are from prev capsule
13    c = X.Cas(exp,new,seq,pid);}
14  *Any number of persistent memory reads
15   and ephemeral memory operations*
16  *Writes to private persistent memory are
17   allowed under certain conditions*
18  capsule_boundary()
```

First note that if a fault occurred, then the capsule never uses any of the local variables before overwriting them. Therefore, its execution does not depend on local values from previous capsules. This includes the sequence number for the capsule, which must have been written in persistent memory before the capsule started, and is therefore the same in all repetitions of the capsule.

Note that in all but the first (partial) run of the capsule, the `fault()` function must return true. Furthermore, note that the code only repeats `X.Cas()` if `checkRecovery` returns false. Due to the correctness of the recovery protocol, this happens only if each earlier operation with this capsule's sequence number has not been executed in a visible way. This means they are either linearized and invisible, or they have not been linearized at all. The partial executions have not been linearized cannot become linearized at any later configuration because X is strictly linearizable. Therefore the `X.Cas()` call is only ever repeated if the previous calls that this capsule made to it were invisible, so all but the last instance of the `X.Cas()` operations executed are invisible. Furthermore, since `X.Cas()` is a strictly linearizable implementation of CAS, the effect of instances together is that of a single CAS. Note that the rest of the capsule is composed of only invisible operations; the recovery of an object is always invisible, as are Reads and local computations. □

Theorem 5.6.2. *A program that uses only CAS-Read, Read-Only, and Single-Instruction capsules is correctly encapsulated, and is a contention-delay-free simulation of its underlying program.*

Proof. Since CAS-Read, Read-Only, and Single-Instruction capsules are all correct (corollary of Theorem 5.6.1), by definition, a program that uses only these capsules is correctly encapsulated. Furthermore, since by Theorem 5.5.1, a program encapsulated with single-instruction capsules only is a constant-contention-delay simulation of its underlying program, and CAS-Read and

Read-Only capsules use strictly less instructions, programs encapsulated with these capsules are also contention-delay free simulations. □

5.6.2 Normalized Data Structures

Timnat and Petrank [274] defined *normalized data structures*. The idea is that the definition captures a large class of lock-free algorithms that all have a similar structure. This structure allows us to reason about this class of algorithms as a whole. In this section, we briefly recap the definition of normalized data structures, and show optimizations that allow converting normalized data structures into persistent ones, with less persistent writes than even our general Low-Computation-Delay Simulator would require. We will show two optimizations; one that works for any normalized data structure, and one that is more efficient, but requires a few more (not-too-restricting) assumptions about the algorithm.

Normalized lock free algorithms use only CAS and Read as their synchronization mechanisms. At a high level, every operation of a normalized algorithm can be split into three parts. The first part, called the *CAS Generator*, takes in the input of the operation, and produces a list of CASs that must be executed to make the operation to take effect. The second part, called the *CAS Executor*, takes in the list of CASs from the generator, and executes them in order, until the first failure, or until it reaches the end of the list. Finally, the *Wrap-Up* examines which CASs were successfully executed by the executor, and determines the return value of the operation, or indicates that the operation should be restarted. Interestingly, the Generator and Wrap-Up methods must be *parallelizable*, intuitively meaning that they do not depend on a thread's local values, and can be executed many times without having lasting effects on the semantics of the data structure.

Optimizations. Our Low-Computation-Delay Simulator works for all concurrent algorithms that access shared objects using only read, write and CAS. In particular, works for normalized data structures. However, we can exploit the additional structure of normalized algorithms to optimize the simulation.

Note that placing capsule boundaries around a parallelizable method yields a correct capsule. This is implied from the ability of parallelizable methods to be repeated without affecting the execution, which is exactly the condition required for capsule correctness. The formal definition of parallelizable methods is slightly different, but a proof that this definition implies capsule correctness appears in [87]. Thus, there is no need to separate the code in parallelizable methods into several capsules. Furthermore, there is also no need to use recoverable CAS for some of the CAS operations performed by parallelizable methods; for normalized data structures, we can simply surround the CAS generator and the Wrap-Up methods in a capsule, and do not need to alter them in any other way.

All that remains now is to discuss the CAS executor, which simply takes in a list of CASs to do, and executes them one by one. No other operations are done in between them. Note that we can convert CAS operations to use the recoverable CAS algorithm, and then many consecutive CASs could be executed in the same capsule, as long as they access different objects. In the case of normalized data structures, however, we do not have the guarantee that the CASs all

access different shared objects. Therefore, we cannot just plug in that capsule construction as is. However, we note another quality of the CAS executor that we can use to our advantage: the executor stops after the first CAS in its list that fails. Translated to the language of persistent algorithm, this means that we do not actually need to remember the return values of each CAS in the list separately; we only need to know the index of the last successful CAS in the list. Fortunately, the recovery operation of the recoverable CAS algorithm actually gives us exactly that; it provides the sequence number of the last CAS operation that succeeded. Therefore, as long as we increment the sequence number by exactly 1 between each CAS call in the executor, then after a crash, we can use the recovery function to know exactly where we left off. We can then continue execution from the next CAS in the list. Note that if the next CAS in the list actually was executed to completion but failed before the crash, there is no harm in repeating it. We simply execute it again, see that it failed, and skip to the end of the executor method.

For a recoverable CAS to work correctly, all CASs to that object must use the recoverable CAS algorithm. Whenever a generator or wrap-up method performs CAS on an object that could also be modified by a CAS-executor, it must use recoverable CAS instead of classic CAS. This is because even though the generator or wrap-up method never needs to recover a CAS's results, it is still important to notify other processes of the success or failure of their last CAS.

We now discuss a method that allows removing the capsule boundary between the executor and the wrap-up. We argue that as long as we can recover the arguments and results of each executor CAS, it is safe restart the execution from the beginning of the executor. Suppose a combined executor plus wrap-up section faults and repeats multiple times, we first argue that as long as the wrap-up part cannot overwrite the notification of any CAS in the cas-list, then in every repetition, the executor returns exactly the same index in the cas-list. Recall that we assume CAS operations are ABA-free in the original program (i.e. the object cannot take on a previous value) and that each process calls CAS using a value it previously read as the expected value. This means that if a CAS operation fails the first time, then the same CAS operation will also fail the second time. Furthermore, since we do not overwrite the result of the executor CASs outside the executor, it can always use the recovery properly to know which CAS in the list was the last to succeed. Therefore the index returned by the CAS-executor will be the same across all repetitions. This means that the executor plus wrap-up capsule basically behaves as if there were a capsule boundary between the two methods. Since the wrap-up method is parallelizable, we know this capsule is correct.

To remove the capsule immediately after the executor, we need to ensure that the wrap-up does not corrupt the ability of the recoverable CAS to tell whether the most recent *executor CAS* on each object succeeded. If the wrap-up does not access any CAS location accessed by the executor, this property is guaranteed. However, if there is a CAS in the wrap-up that accesses the same location as some CAS in the executor, we can still ensure that we can recover. Let C_w be such a CAS in the wrap-up executed by process p. Note that C_w never needs to use the recovery function for itself; since the wrap-up is parallelizable, it is always safe to repeat C_w after a crash. Therefore, when C_w is executed using a recoverable CAS, it can leave out its own ID and sequence number, so that other processes do not notify p. Thus, the previous notification that p received (i.e. a notification about p's executor CAS on the same object) remains intact.

Notice that if we have two parallelizable methods, A and B, next to each other, we can actually put them in a single capsule as long as the inputs to A and B are the same whenever the

capsule restarts. Since A and B have the same inputs, we know by parallelizability that A and B each appear to execute once regardless of how many times the capsule restarts. Also A must appear to finish before B because there was a completed execution of A before any invocation of B. Therefore, this capsule appears to have executed only once.

So, we can avoid an additional capsule boundary between the current iteration's wrap up method and the next iteration's generator method, as long as we now use the same notification trick in the CASs of the generator as well. So as long as there are capsule boundaries before and after each call to a normalized operation, we only need one capsule boundaries in each iteration of the main loop: only before the executor. We call this simulation the *Persistent Normalized Simulator*. The details of our encapsulation are shown in Algorithm 5.4. The results of this section are summarized in Theorem 5.6.3.

Theorem 5.6.3. *Any normalized data structure N can be simulated in a persistent manner with constant-contention-delay using one capsule boundary per repetition of the operation.*

Algorithm 5.4: Persistent Normalized Simulator

```
1  result_type NormalizedOperation(arg_type input) {
2    do {
3      cas_list = CAS_Generator(input);
4      capsule_boundary();
5      if (fault()) {
6        cas_list = read(CAS_list);
7        seq = read(seq); }
8      idx = CAS_Executor(cas_list, seq);
9      ⟨output, repeat⟩ = Wrap_Up(cas_list, idx);
10   } while(repeat == true)
11   return output; }

13 int CAS_Executor(list CASs, int seq) {
14   //CASs is list of tuples ⟨obj, exp, new⟩
15   bool faulted = fault();
16   bool done = false;
17   for (i = 0; i < CASs.size(); i++) {
18     if (faulted) {
19       done = check_recovery(CASs[i].obj, seq, p); }
20     if (!done) {
21       if (!RCAS(CASs[i])) return i; }
22     seq++;  }
23   return CASs.size(); }
```

5.7 Practical Concerns

Flushes and Fences. In this chapter, we've assumed that there is a way to *persist* variables on ephemeral memory by copying them over to persistent memory. On real machines, this can be done using *flush* and *fence* instructions. At a high level, a flush instruction writes out a given cache line to memory. However, most flush instructions do not block until the cache line reaches

memory. Thus, even if a cache line is flushed before a crash, it is possible that the flush instruction did not complete, and the cache line's data could still be lost. To prevent this, one can execute a *fence* instruction, which blocks the execution until after any flushes invoked before it complete.

Implementing Capsules in Real Programs. When interpreting real program code in terms of our model, we assume that all stack-allocated local variables, including the program counter, are updated in ephemeral memory, and that heap-allocated variables are created and updated directly on persistent memory. That is, the ephemeral variables in our model map to stack-allocated variables, and we add to each stack frame the program counter at the last capsule boundary. We therefore keep a copy of the stack in persistent memory, using explicit flush and fence instructions to ensure that the variables get written on memory. We use a doubling trick similar to the one discussed in Section 5.4.1 to handle stack-allocated variables with write-after-read conflicts.

Instead of using write buffers to only duplicate the variables that have write-after-read conflicts, another possibility is to duplicate all stack-allocated variables, that is, have *two* persistent copies of each stack frame, and toggle between the two, copying all values over from one to the other at the end of every capsule. This technique seems wasteful, but is very robust, and works well when the number of variables is small.

Note that from the control flow graph of the program, we can know which variables suffer write-after-read conflicts in a capsule. If these variables stay the same across many capsules, we can optimize the technique a bit, by duplicating only these variables, and having only one copy for the rest of the variables. A validity bit can indicate which copy of the duplicated variables is valid, similarly to how the write buffers work. The validity bit must be flipped after all other changes are made, and flipping it represents the atomic transition between capsules. This technique allows us to use space only as necessary, as the write buffer technique does, but reduces the number of times values are copied over to different locations. If between two capsules the variables with write-after-read conflicts change, we can use the copy-all technique to remap the memory usage to have copies of the correct variables.

Note also that generally, flushes are done at the granularity of a cache line. Thus, if all stack-allocated variables (with the necessary duplications) fit in a single cache line, we can simply ensure that they are all updated before the validity bit is flipped, and only need a single fence to commit the capsule.

Unlike stack-allocated variables, heap-allocated variables are directly written on persistent memory. Therefore, we must make sure that repeated code does not corrupt their values by setting capsule boundaries in between instructions that may harm each other, just like we do for the shared memory instructions.

Shared vs Private Model. In the PPM model used in this work, we assume that the only way for processes to communicate is through persistent memory (i.e., all shared memory is persistent). Furthermore, the volatile memory is explicitly managed, and no automatic flushes occur. This model and similar variants have been used often in the literature [30, 59, 133]. A slightly different model has also been considered in several other works [89, 122, 123, 166]. The main difference between the two models is in how processes communicate. In this other model, sometimes referred to as the shared cache model, processes communicate through objects in *volatile* memory rather than persistent memory. The values in these objects are persisted when the program issues an explicit flush instruction or when a cache line is evicted automatically.

This introduces the possibility that shared memory operations are persisted out-of-order due to implicit cache evictions. Algorithms designed for the shared cache persistent model therefore have to handle this problem in addition to the problems of losing private data discussed in this chapter. The shared model is more faithful to current cache coherent machines, while the private model helps to abstract away machine-specific flushes.

A simple transformation to convert an algorithm for the private cache variant into an algorithm that works in the shared cache variant was presented by Izraelevitz *et al.* in [166]. This transformation applies to programs written in the release-consistency memory model. After every load-acquire, it adds a flush and a fence. Before every store-release, it adds a fence, and after every store and store-release, it adds a flush. When transforming an algorithm from the shared cache model to a model in which cache lines may be automatically evicted, one needs to consider not only shared variables, but also local ones. Inconsistencies can occur in code that might repeat changes to persisted local variables (due to a crash). This can be handled again by avoiding write-after-read conflicts [59], as is discussed for heap-allocated variables in Section 5.6.

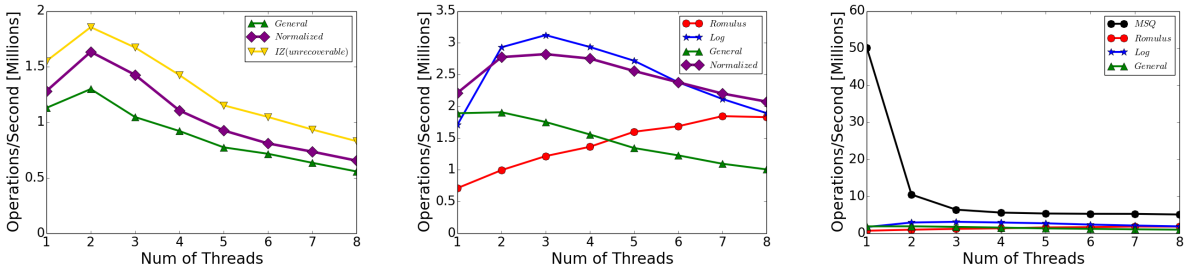
Compiler. Note that we treat shared variables differently from private ones; private heap-allocated variables may be written to many times in a single capsule, but this is disallowed for shared variables. It is thus important that the compiler be able to distinguish between these two types of variables. One way to achieve this is to have the user annotate shared variables. We also need annotations to allow the compiler to determine which of our constructions should be used; for normalized data structures, we assume the user can annotate the generator, executor, and wrap-up sections. We also assume that for simple cases, the compiler can determine if a variable is ever used again. Therefore a capsule boundary only needs to persist the variables that may be used in the future.

Constant Stack Frames. Recall that for the stack-allocated variables, we assume that there is only a constant number of them (around the same as the number of bits in a word) in each stack frame. This is important to be able to atomically update the validity mask of the variables in each capsule boundary, as in Section 5.2.

CAS. Also recall that the recoverable CAS algorithm requires storing not only the value, but also an ID and sequence number in each CAS location. This can be achieved by using a double-word CAS, which is common in modern machines.

Flushes on the Same Cache Line. In a capsule boundary, if all the local variables fit on the same cache line, then we only need one fence for the capsule since the cache line gets flushed all at once in the private model. On the other hand, note that on modern machines with automatic cache evictions, writing all variables on one cache line does not guarantee atomicity, since an eviction can happen part way through updating the cache line. However, we can still assume, as is done in [88], that writes to the same cache line are flushed in the order they are written. Intuitively, this is because on real machines, the following three properties generally hold: (1) total store order (TSO) is preserved, (2) individual words are written atomically, and (3) each cache line is evicted atomically.

Memory mapping. Note that for our algorithms to recover, we assume that after a crash, the each process can always find the memory in which the capsule boundary stored information. This requires persisting the page table. We assume that this is done by the operating system. We further assume that each process is assigned the same virtual address space as it was before the crash. These two assumptions together ensure that each process receives the same *physical*



a) Flushing every instruction

b) Comparing our queues with manual flushes to prior work

c) Comparing persistent queues to original MS Queue

Figure 5.5: Throughput of persistent and concurrent queues under various thead counts. Normalized and General are the result of our transformations.

address space before and after a crash. More details about the virtual to physical mapping for persistent memory is given in [74].

5.8 Experiments

We measured the overhead of our general and normalized data structure transformations by applying them to the lock-free queue of Michael and Scott [232]. We also compare against Romulus [92], a persistent transactional memory framework, and LogQueue [122], a hand-tuned persistent queue. Both Romulus and LogQueue provide durability [166] and detectability [122] in the shared cache model. We use the shared cache model for our experiments because it's closer to the machine that we test on. But this means we need to somehow translate our detectable queues from the private cache model to the shared cache model. We consider two different ways of doing this translation: automatically by Izraelevitz *et al.*'s durability transformation [166] or manually by hand.

We ran our experiments on Amazon's EC2 web service with Intel(R) Xeon(R) Platinum 8124M CPU model (8 physical cores, 2-way hyperthreading, 3GHz and 25MB L3 cache), and 16GB main memory. The operating system is Ubuntu 16.04.5 LTS. Just like in [74, 92, 122], we assume that the cost of flushing on this system will be similar to what we will see in real NVM systems.

All functions were implemented in C++ and compiled using the g++ 5.4.0 with -O3. We only measured the performance on 1-8 threads (each on a separate core), as queues are not a scalable data structure. As in previous work [122, 232], we evaluated the performance with threads that run enqueue-dequeue pairs concurrently. In all the experiments we present, the queue is initiated with 1M nodes; however, we also tested on a nearly-empty queue and verified that the same trends occur. The *flush* operation consists of two instructions: *clflushopt*, and *sfence*. *Clflushopt* has store semantics as far as memory consistency is concerned. It guarantees that previous stores will not be executed after the execution of the *clflushopt*. According to Intel, flushing with *clflushopt* is faster than executing flushes using *clflush* [162]. We also tried using *clwb* instead of *clflushopt* and found no difference in performance. The *sfence* instruction guarantees that the *clflushopt* instruction is globally visible before any following store instruction in program

order becomes globally visible. We omitted some fences when the ordering of the flushes is not important. All the presented results use the *recoverable CAS algorithm* that was proposed by Attiya *et al.* [30]. In our experiments, their algorithm performed slightly better than ours and thus was the one that was presented.

Each of our tests were run for 5 seconds and we report the average throughput over 10 runs. In general, queues that contain less flushes perform better, which is consistent with what we expected.

In our experiments, our goal is to understand the overhead general programs would observe if they were made persistent using various methods. When we run the queue experiments, we keep in mind that these queues should be used within general programs. So, before calling each of the queue operations, the general program has to execute a capsule boundary. This is true for *all* queues that we test, including the LogQueue and Romulus. Therefore, since this additional overhead would be the same for all queues tested, we omit it in our comparative experiments. However, we note that the LogQueue and Romulus produce stand-alone data structures, that maintain more information than our queues do if the initial capsule boundary is removed. This means that in some specific contexts, for example when a few queue operations are executed consecutively, a capsule boundary can be avoided before calling LogQueue or Romulus operations, whereas our constructions still require it. It is possible to store some extra information in our queue constructions to match the properties of the other queues, but this requires some careful manipulations, which are outside of the scope of this work.

Using the Izraelevitz Construction. One general way to automatically achieve correctness in the shared model is to use the construction presented by Izraelevitz *et al.* [166] (described in Section 5.7). Figure 5.5 a) shows the result of applying our transformations along with Izraelevitz's construction to the Michael-Scott queue (MSQ) [232]. To isolate the overhead of our transformations, we also show the performance of a Michael Scott queue with just the Izraelevitz construction. We call this the *IZ* queue and it is an upper bound on how well our transformations can perform. The result of the Low-Computation-Delay Simulator is called *General*. *Normalized* represents the normalized data structure transformation introduced in Section 5.6.2.

As the *General* queue contains more *capsule boundaries* than the *Normalized* queue, we can see that *Normalized* performs 1.25x better when there are 2 running threads, and 1.17x better when there are 8 threads. Without any of the detectability transformations, the *IZ* queue performs better than *Normalized* by 1.13x and 1.26x for 2 and 8 threads respectively.

Competitors. Another way to make our transformed queues correct in the shared model is to add flushes manually. The flushes we add are very similar to those in Friedman *et al.*'s Durable Queue [122]. The difference is that we flush both the head and tail to allow for faster recovery and we omit the return value array; Friedman *et al.* used the return value array to recover return values following a crash, but this functionality is handled by our transformations.

We compared the manual flush version of our transformed queues with the *Log* [122] as well as a queue written using *Romulus* [92]. We chose the *RomulusLR* version as it performed better for every thread count. The results are depicted in Figure 5.5 b). For scalable memory allocation, our transformations use jemalloc and *Romulus* uses its own scalable memory allocator. We ran

Log queue both with and without jemalloc and found that it tends to be faster without jemalloc. The faster set of numbers are reported.

Our *Normalized* queue performs much better than Romulus when the thread count is low, which could partly be because Romulus incurs the extra overhead of implementing a persistent memory allocator and general transactional memory. *Romulus* scales very well and outpaces *General* because it uses flat combining [147], a technique where update transactions are aggregated and processed with a single lock acquisition and release.

When compared to *Log* queue, we found that *Normalized* performs better by 29% on one running thread and by up to 9% on 7-8 threads. The *Log* queue is better by up to 10% on 2-6 threads. We believe this is because *Normalized* performs less overall fences compared to *Log* queue, however, in some places, *Normalized* performs more work in between a read and its corresponding CAS. These instructions bottleneck performance at higher thread counts. With clever cache line usage, it is also possible to reduce *Log* queue enqueues by one flush, but we did not implement this in our experiments. Given that the *Log* queue was tailored to one particular data structure, we were impressed that our *Normalized* automatic construction gets comparable performance. Although we did not measure this experimentally, our transformation also has lower recovery time compared to *Log* queue in situations where the size of the queue is larger than the number of processes. This is because the recovery function of *Log* queue traverses the entire queue starting from the head. In contrast, our recovery function just involves loading the previous capsule and performing the recovery function of a recoverable CAS object. Since we are using Attiya et al's implementation of recoverable CAS [30], recovery time is linear in the number of threads.

Figure 5.5 c) shows how these persistent queues compare to the original MSQ. From the graph, it looks like the cost paid by our transformations to ensure generality and quick recovery is not so much compared to the inevitable cost of persistence.

Part III

Remote Direct Memory Access

Chapter 6

Introduction and Preliminaries

So far in this thesis, we've considered hardware that exists within multicore machines, and modeled it with variants of the shared memory model. In this part of the thesis, we shift our focus to a distributed setting in which the different processes reside on physically separate machines. Usually, such settings are theoretically modeled and studied through the *message passing* model.

The distributed computing community has a dichotomy between shared-memory and message-passing models. Books, courses, and papers explicitly separate these models to present results and algorithms. These models differ based on how processes communicate. As discussed eariler in the thesis, in the shared-memory model, processes can write and read data in a common area of memory. In the message-passing model, processes can send and receive messages to and from each other. The dichotomy in their study is the result of the different motivations for the models, with the shared memory model representing multicore machines, and the message passing model representing systems of machines connected over a network.

In recent years, a technology known as Remote Direct Memory Access (RDMA) has made its way into data centers, earning a spotlight in distributed systems research. RDMA provides the traditional send/receive communication primitives, but also allows a process to directly read-/write remote memory, thereby challenging the assumption that shared memory and message passing cannot coexist in the same system.

Motivated by RDMA, in this part of the thesis, we investigate the benefits of a hybrid model, called the *message-and-memory model* or simply the *M&M model*, where processes can both pass messages and share memory. Such a model can find applications in systems using other emerging technologies as well, such as disaggregated memory [211] and Gen-Z [128]. By using both methods of communication, one can devise a new genre of algorithms that could potentially combine the advantages of shared-memory and message-passing algorithms.

It has been proven that the message-passing and shared-memory models are equivalent [29], by demonstrating that one model can simulate the other. If that is so, what could be the benefit of combining two models that are equivalent? Closer inspection of the equivalence result reveals that it holds only in a certain sense and under some assumptions. In particular, the equivalence is computational: it shows how algorithms for a problem in one model can be translated to the other using an emulation. However, the emulation does not preserve efficiency, fault tolerance, or synchrony (e.g., a timely process in one model can become untimely as it waits for other processes). In fact, we find that each model has its own advantages and they benefit algorithms

in different ways.

It is well known that neither the message passing model nor the shared memory model (with atomic read and write operations), can solve consensus deterministically in an asynchronous setting [119]. However, using randomization or assuming partial synchrony allows both models to get around the impossibility. Interestingly, in such a setting, a message passing system can only tolerate a minority of crash failures for solving consensus, whereas under the same randomization or synchrony assumptions, a shared memory system can solve consensus despite an arbitrary number of crashed processes. This is one concrete way in which shared memory is stronger than message passing. Conversely, when considering not only crash failures, but Byzantine failures as well,[1] it is known that a message passing system can tolerate up to one third of the processes in the system being Byzantine, whereas the shared memory model cannot handle any Byzantine failures unless stronger primitives are available. Thus, the message passing model is also stronger than the shared memory model in some ways.

We consider two slightly different settings in which the M&M model is applicable—large and small networks. RDMA memory operations, and more generally, operations in shared memory, do not scale as well as message passing. This is due to several factors, mostly having to do with cache concerns. As discussed earlier in the thesis, operations in a multicore machine may experience significant slowdowns due to contention. With RDMA operations, the cause is different; the network interface card (NIC) of a machine must keep information about each open RDMA connection on its cache, leading to frequent cache misses if a single machine in the network maintains many RDMA connections. Regardless of the cause, poor scalability remains a reality across all known technologies that allow shared memory communication. Thus, in networks with many processes, it is impractical to have all-to-all shared memory communication. We therefore distinguish between large networks, in which we must diligently choose which RDMA connections to open, and small networks, in which we can allow all-to-all RDMA connections.

For large networks, we consider the problem of crash-tolerant consensus, and show that we can use shared memory to enhance the fault tolerance of a message passing system. In fact, we show that the fault tolerance achievable in such a network is related to the topology of the graph of shared memory connections in the network.

We then consider small networks, in which all-to-all RDMA connections are practical. Our goal is to understand the full power that technologies such as RDMA can provide. We therefore bring in more RDMA-specific features to our model. In particular, we consider RDMA's ability to give and dynamically change specific access permissions to memory, as well as the possibility of experiencing *memory failures*. Under this RDMA-specific version of the M&M model, we consider implementing efficient and fault tolerant consensus under two types of process failures: crash-stop and Byzantine. For Byzantine failures, we give an algorithm that (a) requires only $n \geq 2f_P + 1$ processes (where f_P is the maximum number of faulty processes) and (b) decides in two network delays in the common case. This is an improvement in fault tolerance over both the classic, permissionless, read/write shared memory model and the message passing model, and is as efficient as is possible in the message passing model. With crash failures, we give

[1] A *Byzantine* process may arbitrarily deviate from its protocol, and is assumed to collude with the adversarial scheduler to try to make the algorithm fail.

the first algorithm for consensus that requires only $n \geq f_P + 1$ processes and decides in two network delays in the common case. With both Byzantine or crash failures, our algorithms can also tolerate crashes of memory—only $m \geq 2f_M + 1$ memories are required, where f_M is the maximum number of faulty memories. Furthermore, with crash failures, we improve resilience further, to tolerate crashes of a minority of the combined set of memories and processes. Thus we show that, aside from the crash-fault tolerance already studied in the large network setting, RDMA can also help improve Byzantine-fault tolerance and efficiency of consensus algorithms.

Finally, to evaluate the practicality of our model, in Chapter 9 we use insights from the crash-tolerant small-network consensus algorithm presented in Chapter 8 to develop an RDMA-based *state machine replication* (SMR) system. SMR helps increase the availability of a system by replicating the state machine of an application across several servers, thereby allowing the system to remain available despite a server crash. The core engine of an SMR system makes use of a consensus algorithm to ensure the replicas all have the same state. We employ the consensus algorithm developed from our theoretical model, extending it to support a full SMR system. The resulting system, called *Mu*, can replicate requests in only $1.3\mu s$, outperforming other state-of-the-art SMR systems by up to $6\times$ in the common case, and over an order of magnitude when there are server failures. This therefore shows that our theoretical study of RDMA enabled us to make an impact in practice.

6.1 RDMA in Practice

To keep the discussion of our RDMA model grounded in reality, we now briefly discuss its capabilities in practice. We highlight the features that we model in this thesis and later make use of in our SMR implementation.

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

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

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

6.2 The M&M Model

In this part of the thesis, we model the features of Remote Direct Memory Access (RDMA) and similar technologies, like GenZ and disaggregated memory, that allow for both shared memory and message passing communication in the same network. We present the most general form of our model, called the message-and-memory model, or the M&M model, in this section, and later extend it to model RDMA-specific features in Chapter 8. We begin by assuming a shared memory system, like in the rest of the thesis. However, this time, the system represents processes residing on different physical machines. We therefore make a few different assumptions about the primitives that processes can use.

First, for algorithms in this part of the thesis, instead of assuming that any number of processes can fail, we assume an upper bound f_p for the number of processes that can crash during the execution.

Secondly, we assume that processes use only atomic read and write registers in shared memory, and disallow FAA and CAS. In practice, some hardware for communication on different machines, like RDMA, does allow FAA and CAS, but these primitive are notoriously slow [172, 281]. We therefore focus on what can be achieved without it.[2]

Shared-Memory Layout. To represent hardware limitations on how many shared-memory connections can be open on each NIC at once [106, 170, 173, 275], we restrict each shared-memory location to only be shared by a subset of the processes. We define the *shared-memory domain* $\mathcal{S}$ as a set of process subsets; intuitively, $\mathcal{S}$ determines what subsets of processes can share memory. More precisely, for each set $S \in \mathcal{S}$, the model permits having any number of registers shared among processes in S. In general, $\mathcal{S}$ can be arbitrary. However, in practice memory sharing is simpler, as the hardware technology naturally imposes a structure on $\mathcal{S}$: for example, a process might be able to share memory only with processes that connect to it over the underlying hardware. We say that $\mathcal{S}$ is *uniform* if it can be represented by an undirected graph G_{SM} of processes, such that registers can be shared by a process and its neighbors in G_{SM}; intuitively, G_{SM} is the graph of connections of the underlying hardware that implements the shared memory. Formally, G_{SM} is a graph $G_{SM} = (\Pi, E_{SM})$ and the sets in $\mathcal{S}$ are exactly the sets consisting of a process p and its neighbors in G_{SM}. That is, if we let $S_p = \{p\} \cup \{q : (p, q) \in E_{SM}\}$ then $\mathcal{S} = \{S_p : p \in \Pi\}$. For a uniform $\mathcal{S}$, we say that G_{SM} is its *shared-memory graph*.

We are interested in the uniform model, and all our results work with the graph G_{SM}. The broader model based on $\mathcal{S}$ is provided to allow for future theoretical work and potential new hardware platforms. Note that, while the model does not constrain the number or size of registers that can be shared, algorithms may choose to reduce their shared-memory usage for efficiency.

In systems with few processes (e.g., in the tens), G_{SM} could be a fully connected graph, but systems with lots of processes may have to limit the maximum degree of G_{SM} (e.g., limit the connections over the hardware).

[2]For the algorithms in the first chapter in this part of the thesis, using CAS and FAA would not change the results, as we use shared memory simply as a black-box to solve consensus tolerating $n - 1$ failures.

Message Passing. In addition to shared memory, we assume that processes may communicate using *message passing*. Processes can send messages over directed links. The link from p to q, denoted $p \to q$, is an *output link* of p and an *input link* of q. Each link $p \to q$ satisfies the following property in every run:

- *[Integrity]:* If q receives m from p k times then p previously sent m to q at least k times.

Some links may satisfy additional properties which are described next. We consider two types of links: reliable and fair lossy. Roughly, a reliable link does not drop messages, while a fair lossy link may drop messages but ensures that if a process sends a message repeatedly then it is eventually received.

More precisely, we say that a link $p \to q$ is *reliable* if it satisfies Integrity and the following property:

- *[No loss]*: If p sends a message m to q and q is correct then eventually q receives m from p.

We say that a link $p \to q$ is *fair lossy* if it satisfies Integrity and the following property:

- *[Fair loss]*: If p sends a message m infinitely often to q and q is correct, then q receives m infinitely often from p.

In this thesis, we always consider a fully-connected message-passing network, that is, for any two processes $p \neq q$, there is a link from p to q.

6.2.1 Problem Definitions

We now formally define the *consensus* problem considered in this part of the thesis.

Consensus Problem. In the consensus problem, each process begins with an input value $v \in \{0, 1\}$ which it *proposes*, and it must make an irrevocable *decision* on an output value in the end.

With crash failures, we require the following properties:

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

We expect Agreement and Validity to hold in an asynchronous system, while Termination requires standard additional assumptions (partial synchrony, failure detection, etc). We also allow randomized solutions, where the above Termination property must hold with probability 1 under a *strong adversary*—one that can schedule processes based on their current state and past history.

In a Byzantine setting, the requirements must be adjusted a little. In particular, we cannot require that all agents that decide a value agree, since Byzantine agents may decide arbitrary values. Similarly, we cannot require that Byzantine agents output valid decision values. We consider weak Byzantine agreement [192], with the following properties:

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

- **Termination.** Eventually all correct processes decide.

A *consensus object* is a shared-memory object with one operation, $propose(v)$, which takes a value v and returns the first value that was proposed to the object. The process that proposed the first value to the consensus object is the *winner* of that consensus object.

Chapter 7

Large Networks

We begin the study of RDMA with a simple model that combines classic shared memory with classic message passing. This chapter is based on results presented in [12].

Shared-memory systems have worse scalability than message-passing systems due to hardware limitations. For example, a typical shared-memory system today has tens to thousands of processes, while message-passing systems can be much larger (e.g., map-reduce systems with tens of thousands of hosts [102], peer-to-peer systems with hundreds of thousands of hosts, the SMTP email system and DNS with hundreds of millions of hosts, etc). This limitation is also apparent in RDMA networks. If a single node in the network maintains many open RDMA connections, significant slowdowns can be observed for RDMA operations on that node.

To model efficient networks with many nodes, we thus model an RDMA-enabled system as a system in which all machines can communicate with each other over message passing (all-to-all links), but only subsets of the system can share memory. We assume that the shared memory does not fail, as in the pure shared-memory model. This assumption can be supported by the hardware: with RDMA, the shared memory can be registered with the kernel so that it remains accessible after processes crash. Even other technologies, like disaggregated memory, can similarly preserve memory accesses after process crashes. The formal model is described in Section 6.2.

Under this model, we consider the problem of crash-tolerant consensus, and show that even with a few shared memory connections, consensus can be solved with higher fault tolerance than is achievable with message passing alone. It is known that consensus cannot be solved deterministically in an asynchronous system subject to failures, even if processes can only fail by crashing and at most one process may fail; this is true in both message-passing and shared-memory models [119, 220]. We thus consider asynchronous systems where processes can toss coins. Then, in a shared-memory system, consensus can be solved (with probability 1) with up to $n-1$ crash failures [2] (n is the number of processes in the system), whereas in a message-passing system, a consensus algorithm can tolerate at most $\lfloor (n-1)/2 \rfloor$ crash failures [51]. We show that the M&M model can strike a balance between shared memory and message passing.

We define an undirected *shared-memory graph*, whose nodes are processes and there is an edge between processes p and q if they share a memory location. First note that if the shared-memory graph G_{SM} is fully connected then any fault-tolerant shared-memory algorithm also works in the M&M model—the algorithm simply never sends messages. Thus, there are algo-

rithms in the M&M model that can tolerate up to $n-1$ crash failures. However, as discussed above, in a large system, it is impractical to connect all processes over shared memory. When fewer processes can share memory, we show that the M&M model provides a range of choices, where the fault tolerance increases as we improve the shared-memory graph. Specifically, we present an algorithm for the M&M model that tolerates anywhere between $\lfloor (n-1)/2 \rfloor$ and $n-1$ crash failures, depending on the topology of the shared-memory graph.

In particular, the hardware limitations on scalability translate to limitations on the degree d of the shared memory graph [106, 170]. Our algorithm employs expander graphs to tolerate a *majority* of crash failures—up to $f < (1-\frac{1}{2(1+h)})n$ of them—in a system with n processes, where h is the expansion of the graph as measured by the *vertex expansion ratio*. Roughly, this ratio indicates by how much a set of vertices expands each time we add their neighbors to the set. The higher the expansion, the more failures the algorithm can tolerate. Our algorithm is a simulation of a pure message-passing consensus algorithm that requires a majority of correct processes, without having that majority in reality. To do that, we use a wait-free shared-memory consensus algorithm among each local neighborhood in the shared-memory graph to emulate a virtual process in the larger message-passing algorithm. This virtual process fails only if all processes in its neighborhood fail. By taking a shared-memory graph with high expansion, we ensure that even with a small d, many processes can fail without affecting a majority of virtual processes. Here, the topology of the shared-memory graph determines the fault tolerance of consensus: graphs with higher expansion allow for higher fault tolerance, because correct processes are adjacent to (and thus can simulate) more processes. We show that this relation is inherent by giving an impossibility result relating graph expansion and fault tolerance.

7.1 Related work

The M&M model is motivated mostly by RDMA [160, 165, 255], but also by similar emerging technologies such as disaggregated memory [211], Gen-Z [128], and OmniPath [163]. These technologies provide remote memory [10] and can be unified under higher-level abstractions [11]. RDMA permits a process to access the memory of a remote host without interrupting the remote processor. It has been widely used in high-performance computing [283] and is now being adopted in modern data centers [137]. Work on RDMA shows how it can improve the performance of important applications, such as key-value storage systems [106, 170, 234], database systems [38, 284], distributed file systems [221], and more [107, 267, 271]. Recent work uses RDMA to improve performance of consensus [250, 279] assuming a majority of processes are correct. Disaggregated memory separates compute and memory, and connects them using a fast network; prior work proposes new architectures for disaggregated memory [24, 95, 240] and studies the network [142] and system [10, 212] requirements for a practical implementation. Gen-Z and OmniPath are commercial technologies under development that offer memory semantics and low-latency access to remote data.

The shared-memory and the message-passing models are well studied in academic research, and have been compared under both theoretical and practical considerations [68, 75, 184, 200]. The two models have been shown to be computationally equivalent [29], though for efficiency, simplicity, or hardware availability, one might prefer one model over the other. For instance,

Barrelfish [39] uses message passing to improve performance on a shared-memory multicore machine [99]. Conversely, distributed shared-memory systems [18, 55, 259] offer the abstraction of shared memory on top of a message-passing system. Recent work improves the performance of such systems using RDMA [179, 239]. Integrating message passing and shared memory in hardware has been explored in the MIT Alewife machine [190]. Our work differs in that (1) we propose a new abstract distributed computing model, which encapsulates low-latency remote access technologies, such as RDMA and disaggregated memory, and (2) we show that this model can improve the robustness of algorithms, rather than the performance or simplicity of applications.

Consensus is a fundamental problem in distributed computing. Following the well-known FLP result [119] showing that it cannot be solved in an asynchronous crash-prone message-passing system, much work has focused on getting around the impossibility by using randomization [25, 28, 51], partial synchrony [104, 109], or unreliable failure detectors [76, 77].

Expander graphs are graphs that are sparse, yet well-connected. They are well-studied and have applications in many areas of computer science, including distributed computing [91, 282]. In this chapter, we show that the fault tolerance of the M&M model is tightly coupled with the expansion of its shared-memory connections, highlighting another problem in which expander graphs apply.

7.2 Consensus Algorithm

We now present an algorithm in the M&M model that improves the fault tolerance of message-passing systems using shared memory connections. We first discuss the algorithm itself, which works regardless of how many or which shared memory connections are available, and then discuss the best topology of shared memory connections to choose to optimize scalability and fault tolerance. We then give an impossibility result about the fault tolerance of consensus in the M&M model (Section 7.4).

7.2.1 Algorithm

The algorithm for the M&M model is based on Ben-Or's randomized algorithm [51], which can tolerate up to $f < n/2$ process crashes in the message-passing model. This is one of the simplest consensus algorithms, but not the most efficient one. Our goal here is to show feasibility; designing more efficient algorithms for the M&M model is future work.

Ben-Or's Algorithm. In Ben-Or's algorithm, each process has an estimate of the decision value, which starts with the process's initial value. The algorithm proceeds in rounds, each with two phases. In the first phase (phase R), each process p sends its current estimate to all processes, waits to receive at least $n-f$ messages, and checks if more than $n/2$ messages have the same value v. If so, p sends this value to all processes in the second phase (phase P). Otherwise, p sends a special value '?' to all processes in the second phase. Process p then waits to receive at least $n-f$ messages. If at least $f+1$ of them have the same non-'?' value, p decides on this value. If at least one of them is a non-'?' value, p changes its estimate to that value. Otherwise, p changes its estimate to a random bit.

Ben-Or's algorithm satisfies the validity and uniform agreement properties of consensus (Section 5.2), and it satisfies the termination property with probability 1 if a majority of the processes are correct [9].

Simulating Ben-Or's Algorithm in the M&M model. We modify Ben-Or's algorithm, so that correct processes simulate the actions of their neighbors in the shared-memory graph G_{SM}. The idea of the simulation is simple: when sending a message in any phase, process p sends not only its own value, but also the value that its neighbors are supposed to send. That is, p ensures that its neighbors progress at least as much as it does. To do so, for each neighbor q, p reaches agreement with q and q's neighbors on what q's message should be. Then, p sends a message of the form $(phase, round, [\langle q, val\rangle : q \in neighbors(p)])$, where $phase$ is a phase (either R or P), $round$ is a round number, and the last entry is an array with a tuple for each neighbor q indicating the agreed value of q's message. We say that the message *represents* each of the processes whose ids appear in the tuple. For a set of such messages, we say that the messages represent the union of the processes that are represented by each message separately.

To reach agreement on the message of a neighbor q, there are two arrays of consensus objects, one array for each phase, indexed by q and the round. All of q's neighbors use the same consensus object to determine what q's message might be for a given phase and round. Process q and its neighbors propose their own value to that consensus object. The consensus objects themselves are implemented using known wait-free randomized shared-memory algorithms [25, 28], which work in the M&M model because neighbors in G_{SM} share memory.

We call this algorithm the *Hybrid Ben-Or* or *HBO* algorithm. Algorithm 7.1 shows the pseudocode. There, processes do not terminate after deciding, but it is easy to modify the algorithm so that they do. This algorithm always satisfies the safety properties of consensus, irrespective of the number of crash failures:

Theorem 7.2.1. *The HBO algorithm in Algorithm 7.1 satisfies the Validity and Uniform Agreement properties of consensus in the M&M model with reliable links.*

7.2.2 Correctness.

The correctness of our algorithm relies on that of Ben-Or's. We show that, intuitively, every execution of our algorithm corresponds to some execution of Ben-Or's algorithm. By mapping executions of our algorithm to those of Ben-Or's, we thus show that the possible behaviors of our algorithm are a subset of the behaviors exhibited by Ben-Or's algorithm, and thus all lead to correct solutions to consensus.

We begin with a series of definitions. First, instead of counting full rounds, for the purpose of this proof, we count each *phase* separately. That is, we say that a message m is in *phase* j if m is of the form (R, k, M) where $j = 2k$, or (P, k, M) where $j = 2k + 1$. The *state* of a process comprises of (a) its local values, (b) its current phase, and (c) the messages it has sent and received since the beginning of the algorithm's execution. A process p may take one of two *actions* per phase; p may deterministically update its local estimate to a value $v \in \{0, 1, ?\}$ (called a *det-update*), or p may update its local estimate by flipping a coin (called a *rand-update*). After taking an action, p always sends a message with its new estimate to all processes. A *protocol*

Algorithm 7.1: Hybrid Ben-Or (HBO) consensus algorithm

```
1  Shared objects:
2    RVals[p, i]: consensus object accessible by {p} ∪ neighbors(p),
3    ∀p ∈ Π, ∀i ∈ {1, 2, ...}
4      PVals[p, i]: consensus object accessible by {p} ∪ neighbors(p),
5        ∀p ∈ Π, ∀i ∈ {1, 2, ...}

7  Code for process $p$:
8    procedure Consensus(v_p)
9      message = []
10     k = 1
11     for q ∈ {p} ∪ neighbors(p) do
12       message[q] = ⟨q, RVals[q, k].propose(v_p)⟩
13     while true
14       send (R, k, message) to all
15       wait for messages of the form (R, k, ∗) representing more than n/2
             ↪ processes
16       if received more than n/2 tuples with different ids and the same
             ↪ value v
17         for q ∈ {p} ∪ neighbors(p) do
18           message[q] = ⟨q, PVals[q, k].propose(v)⟩
19       else  for q ∈ {p} ∪ neighbors(p)  do
20           message[q] = ⟨q, PVals[q, k].propose(?)⟩
21       send (P, k, message) to all
22       wait for messages of the form (P, k, ∗) representing more than n/2
             ↪ processes
23       if received more than n/2 tuples with different ids and the same
             ↪ value v ≠ ?
24         decide(v)
25       k = k + 1
26       if at least one tuple has value v ≠ ?
27         for q ∈ {p} ∪ neighbors(p) do
28           message[q] = ⟨q, RVals[q, k].propose(v)⟩
29       else for q ∈ {p} ∪ neighbors(p) do
30           v = 0 or 1 randomly
31           message[q] = ⟨q, RVals[q, k].propose(v)⟩
```

step of a process p is an action taken by p, followed by sending a message. A protocol step by p in which p took action a and sent message m is denoted as (p, a, m). The first protocol step of every process p is defined as a det-update in which p sends its initial value. In our HBO algorithm, process p may also take *helping steps* to help its neighbors perform a protocol step.

Observation 7.2.2. *In Ben-Or's algorithm, two processes with states s and s' take the same action as long as they are in the same phase, and all messages received since the last protocol step they took were the same. Furthermore, if the action taken is a det-update, then their next message is the same.*

This observation is easy to see from a quick inspection of Ben-Or's algorithm. It simply states that processes act deterministically based only on the messages received in the last phase; the only source of non-determinism is the random coin flip. In particular, a process's past history and its previous estimate of the value have no effect on its next action. We thus define a *contracted state* of a process as the subset of its state containing only its current phase and the messages received since the process last took a protocol step.

Recall that an *execution* E of an algorithm is a sequence of protocol steps taken by processes in the system, in which, for any process p, the subsequence containing only steps of p is consistent with the deterministic actions p can take. An *extension* of an execution E is a concatenation of E with another execution E' of the same algorithm, in which the first protocol step of each process p in E' is consistent with the next protocol step p could take after its last step in E. We denote by $p_{init(E)}$ process p's initial (input) value in execution E.

Note that the messages sent in our HBO algorithm are not the same as the messages sent in Ben-Or's algorithm. To be able to compare executions of the two algorithms, we define a mapping of an execution of the HBO algorithm to a sequence of protocol steps of Ben-Or's algorithm. Given an execution E of the HBO algorithm, E's *flattened execution*, E_f, is the sequence of protocol steps where every protocol step $(p, a, (phase, round, [(q, v_q)|q \in N(p)]))$ in E is substituted with the sequence of protocol steps $(q, a_q, (phase, round, v_q))$ for $q \in N(p)$, where a_q is the action taken by the process that determined q's value for that phase (won its consensus). Repeated protocol steps are deleted.

Intuitively, this flattening just corresponds to an execution in which a process p *receives a message from q in phase j* if p receives some message in phase j that contains the tuple (q, v) for some $v \in \{0, 1, ?\}$. Similarly, *q sends a message in phase j* if there is a message sent by some process in phase j that contains the tuple (q, v) for some value v. Furthermore, redundant messages are ignored in the flattened execution.

We say that an execution E of the HBO algorithm and an execution E' of Ben-Or's algorithm are *equivalent* if the flattened execution of E, E_f, is equal to E'.

We now turn to the actual correctness proof. First, we show that although multiple processes may take a protocol step on behalf of p, they all take the same step, and thus there are no inconsistencies in p's representation.

Lemma 7.2.3. *For any execution E of the HBO algorithm, E_f cannot contain more than one protocol step per process per phase.*

Proof. It is easy to see that this lemma holds, since the value sent for each process p in phase j is always determined by a proposal to the same consensus object. Every proposal to the same

consensus object returns the same value. Thus, every time p is represented in a message in phase j in E, it is the same message. By the definition of E_f, each protocol step appears only once. □

We note that an execution of the HBO algorithm could be equivalent to an execution of Ben-Or's algorithm that has a *different set of input values*. This difference is due to the fact that in the HBO algorithm, a process can impose its own initial value on its neighbors via the first consensus object of each process. However, the decision that the HBO algorithm arrives at in the end is always a valid decision *for the original inputs*, since consensus is a colorless task[1].

Observation 7.2.4. *In the Hybrid Ben-Or Algorithm, processes may adopt the input values of their neighbors. This does not affect the validity of the decision value.*

Lemma 7.2.5. *For any execution E of our HBO algorithm, there exists an execution E' of Ben-Or's algorithm such that E and E' are equivalent. E and E' do not necessarily have the same initial values for all processes, but if a process p has input v in E', then at least one of its neighbors has input v in E.*

Proof. The proof is by induction on the number of protocol steps in the execution E_f.

BASE OF THE INDUCTION: Consider the empty prefix of E_f. This is equivalent to any empty execution E' of the original algorithm.

STEP OF THE INDUCTION: Assume that for any prefix S of E_f of length at most k, there is an equivalent execution E' of Ben-Or's algorithm. We now show that there is an extension of E' in which the next protocol step taken is the same one taken in E_f.

Let p be the next process to take a protocol step in E_f, and let j be p's phase for this step. By Lemma 7.2.3, p hasn't yet taken a protocol step in phase j. This means that, since by the induction hypothesis S and E' are equal, there is an extension of E' in which p takes a protocol step in this phase. In order for p to take a protocol step, consensus has to have been reached for the step it should take. Consider the process q that proposed the winning value in p's consensus object for this phase. We consider two cases: when $j = 0$ (the first phase), and when $j \neq 0$.

CASE 1: $j = 0$. If this is the first phase, then q imposes its own initial value for p. Note that there is an extension of E' in which this is p's first step; however, it is an execution in which $p_{init(E')} = q_{init(E)}$. Since q is p's neighbor, this satisfies the lemma.

CASE 2: $j \neq 0$. If this is not the first phase, by Algorithm 7.1, q proposed its value after having received at least $n/2 + 1$ messages from other processes in the previous phase. By the induction hypothesis, these same messages have also been sent in E' (since so far the protocol steps taken have been the same in both executions). Furthermore, since all messages in both the HBO algorithm and Ben-Or's algorithm are sent to all processes, then we can let these $n/2 + 1$ messages be the ones received by p in this phase in the extension of E'. Thus, by Observation 7.2.2, p takes the same action as q would in this extension of E'. This is what happens in the HBO algorithm as well. Note that if p's action is a rand-update, there is an extension of E' in which the result of p's coin flip is the same as the result q proposed for p in the HBO algorithm. Therefore, there is an extension of E' in which the next protocol step by p is the same as the one taken by p in E_f. □

[1] A *colorless task* is one in which processes may adopt the input value of other processes without affecting the validity of the output [153].

Lemma 7.2.5, Observation 7.2.4 and the correctness of Ben-Or's algorithm immediately imply the following theorem.

Theorem 7.2.6. *Algorithm 7.1 satisfies uniform agreement and validity.*

7.2.3 Termination.

For pedagogical reasons, we first show that Algorithm 7.1 terminates under a weak adversary (one that cannot see the local and shared-memory values of processes). Later, we show that the algorithm in fact terminates under a strong adversary as well. Note that in any round, at most one non-'?' value can be proposed. This is because there need to be a strict majority of the processes that reported that value in the report phase of that round. Furthermore, if all processes that flip a coin in some round k obtain the same value as the one proposed in that round (if there is such a value), then all processes start round $k+1$ with the same value, and they all decide at the end of round $k+1$. Note that this scenario has a non-zero probability of happening, since all coin flips are independent of one another, and the adversary cannot alter the majority value to be the opposite of the value flipped, since it cannot observe the local values of the processes. Thus, given enough rounds, the algorithm must terminate eventually, under a weak adversary.

We now turn to arguing about termination under a strong adversary. Ben-Or's algorithm terminates even under a strong adversary, provided that there is a majority of correct processes [9]. The proof of termination is non-trivial, and requires arguing about two rounds at a time. The idea of the proof is to define a 'lucky epoch'; two consecutive rounds in which each coin flip returns the value that will be best in that situation to lead to termination (called 'FavorableToss' in [9]). The main idea is that depending on the current majority opinion, the best coin flip can be different at different times in the execution. Thus, FavorableToss takes in both a round number, r, and a time t at which the coin is flipped. Aguilera and Toueg then show that if, for all processes in two consecutive rounds, any query of the random number generator (r.n.g.) at any time t returns $FavorableToss(r,t)$, then the algorithm terminates.

With the HBO algorithm, the time at which a coin for a particular process is flipped is not well defined; to establish its value for the next round, a process must reach consensus with all of its neighbors on its value. Each of its neighbors may separately flip a coin, and then compete in the consensus protocol to have its value chosen. This gives the adversary more power, as it can effectively choose which neighbor will win. We now show that despite this extra power, the HBO algorithm still terminates with probability 1, even under a strong adversary. We do so by closely following the proof of Aguilera and Toueg for the original Ben-Or algorithm, with a few changes clearly defining the time at which a process is said to flip a coin in a given round, and incorporating this into the definition of a FavorableToss.

Definition 7.2.7. *For every process p and round r, the* time, *t, at which p is said to have tossed its coin for round r is the earliest time in the execution at which some process in the neighborhood of p queries the r.n.g. to obtain a value to propose for p.*

We thus alter the definition of FavorableToss as shown in Algorithm 7.2. The algorithm is taken from [9], and the lines in red are the ones added for the proof of the HBO algorithm. Here, τ_k is the first time in which some process receives at least $n-f$ proposals in round k.

We also slightly modify the terminology used by Aguilera and Toueg, and say that a process

Algorithm 7.2: FavorableToss specification

```
1  FavorableToss(k, t, p) {
2    if there is a time t' < t such that FavorableToss(k,t',p) was called {
3      return FavorableToss(k, t', p);
4    }
5    if at time t there  is a majority value v in round k {
6      return k mod 2;
7    }
8    if v is a majority value in round k+1 by time t { //t ≥ τ_{k+1}
9      return v;
10   }
11   return k+1 mod 2
12 }
```

R-gets a value in round k if the winner of its consensus object for the *propose* phase in that round got its value by flipping a coin. Otherwise, we say that the process *D-gets* its value in that round.

The rest of the proof is identical to the proof given by Aguilera and Toueg. The key insight that allows us to use the old proof is that, with these new definitions, there is a well defined point in time at which a process p's value for round k is determined, and this point in time is always strictly before p sends its first message in round k. Note that the probability for two rounds in a row to be lucky is now only 2^{-2dn}, since each process's value is determined by at most d coin flips. However, this still suffices to show that the probability that we will eventually have a lucky epoch is 1.

7.3 Shared-Memory Expanders

In this section, we consider the fault tolerance of the HBO algorithm: how many crash failures can it tolerate while ensuring that processes decide. In the algorithm, correct processes represent their neighbors in G_{SM}, so the fault tolerance depends on G_{SM} and how many neighbors correct processes have. We show that, by choosing G_{SM} to be a graph with *high expansion*, we obtain the best trade-off between maintaining low degree and achieving high fault tolerance. Having a low degree is important because the degree indicates the number of connections that a process must have to establish a shared memory, and that number is limited by the hardware (Section 5.2).

Roughly, expander graphs are graphs with the property that every sufficiently small set of vertices has many neighbors. To define these graphs more precisely, we follow the survey by Hoory, Linial and Wigderson [157]; we refer the reader to this survey for a detailed treatment.

Definition 7.3.1. *Let $G = (V, E)$ be an undirected graph.*

1. *The* vertex boundary *of a set $S \subseteq V$ is*

$$\delta S = \{u \in V : \{u, v\} \in E, v \in S\} \setminus S.$$

2. *The* vertex expansion ratio *of G, denoted $h(G)$, is defined as*

$$h(G) = \min_{S \subseteq V : |S| \leq |V|/2} \frac{|\delta S|}{|S|}$$

Intuitively, the higher the vertex expansion ratio, the larger the vertex boundary and the better connected the graph.

To apply this definition to the fault tolerance of HBO, consider a system with shared-memory graph G_{SM} and vertex expansion ratio $h(G_{SM})$, where up to f processes may crash. The set of vertices of interest to us is the set C of correct processes. If C has many neighbors, then it can simulate many extra processes in HBO. The adversary may pick *any* set of at least $n - f$ processes to be correct; regardless of the set it picks, that set has a vertex boundary of at least $(n-f) \cdot h(G_{SM})$.

The fault tolerance of HBO improves with the vertex expansion ratio of the graph. This is made precise by the following:

Theorem 7.3.2. *Consider the M&M model with shared-memory graph G_{SM} where links are reliable and f processes may crash. The HBO algorithm in Algorithm 7.1 satisfies the Termination property of consensus with probability 1 if $f < (1 - \frac{1}{2(1+h(G_{SM}))}) \cdot n$.*

Proof. Recall that Ben-Or's algorithm requires that $f < n/2$ for termination. The HBO algorithm simulates the correct processes and the processes in their vertex boundary. Given G_{SM} has vertex expansion ratio $h(G_{SM})$, the number of processes simulated by the algorithm is at least $(n-f) \cdot (1+h(G_{SM}))$. Rearranging the terms leads to the theorem. □

In the rest of this section, we give an example of the kind of graphs we can use, by discussing an explicit construction of one family of expander graphs and showing the actual value of $h(G)$ that they yield.

7.3.1 Explicit Construction: Margulis Graphs.

Before presenting a construction that yields graphs with good expansion, we begin with a few general definitions and facts about graphs that will help us analyze expansion properties. Again, we closely follow the survey of Hoory *et al.* [157].

A *d-regular* graph is a graph in which each vertex has degree exactly d. The *Adjacency Matrix* of an n-vertex graph G, denoted $A(G)$, is an $n \times n$ matrix whose entry (u, v) is the number of edges between u and v in G. Since $A(G)$ is real and symmetric for any G, it has n real eigenvalues, which we denote $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$. These eigenvalues are also known as the *spectrum* of the graph G. In d-regular graphs, $\lambda_1 = d$.

The *spectral gap*, i.e., the difference between λ_1 and λ_2, is an important indicator for the expansion of a graph. Denote $\lambda(G) = \max(|\lambda_2|, |\lambda_n|)$. Often $\lambda(G)$ is used instead of using λ_2 when comparing to λ_1. Note that $\lambda_2 \leq \lambda(G)$, so that the spectral gap is at least as large as the difference between λ_1 and $\lambda(G)$. Where the context is clear, we may use λ to mean $\lambda(G)$.

We now consider an alternative definition for the expansion of a graph.

Definition 7.3.3. $\Psi'_v(G, k) = \min_{S \subset V || S| \leq k} \frac{\Gamma(S)}{S}$*, where $\Gamma(S)$ is the neighborhood of S, including S itself.*

Note that this definition is very similar to $h(G)$. The only differences are that (1) $\Psi'_v(G, k)$ is more flexible, in that it allows us to bound our discussion to subsets of size up to k, rather than up to $n/2$, and (2) the fraction is in terms of $\Gamma(S)$ rather than δS. There is a simple conversion between the two: $h(G) \geq \Psi'_v(G, n/2) - 1$.

We are now ready to present the *Margulis Graph* construction.

Definition 7.3.4 (Construction 8.1 in [157]). *Define the following* 8*-regular graph* $G_n = G = (V, E)$ *on the vertex set* $V = \mathbb{Z}_n \times \mathbb{Z}_n$*. Let*

$$T_1 = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}, T_2 = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}, e_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, e_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Each vertex $v = (x, y)$ *is adjacent to the four vertices* $T_1 v$*,* $T_2 v$*,* $T_1 v + e_1$*,* $T_2 v + e_2$*, and the other four neighbors of* v *are obtained by the four inverse transformations. Note that all calculations are* $\mod n$ *and this is an* 8*-regular undirected graph that may have multiple edges and self loops.*

Theorem 7.3.5 (Theorem 8.2 in [157], due to Gaber and Galil [125]). *The graph* G_n *satisfies* $\lambda(G_n) \leq 5\sqrt{2}$ *for every positive integer* n*.*

An (n, d, α)*-graph* is a d-regular graph on n vertices, in which $\lambda \leq \alpha d$. Thus, we know that Margulis graphs are (n, d, α)-graphs for $d = 8$ and $\alpha \geq 5\sqrt{2}/8$.

Theorem 7.3.6 (Theorem 4.15 in [157], due to Tanner [272]). *An* (n, d, α)*-graph* G *satisfies*

$$\Psi'_v(G, \rho n) \geq \frac{1}{\rho(1 - \alpha^2) + \alpha^2}$$

for all $\rho > 0$*.*

If we plug in $\rho = 1/2$ to the above theorem, we get that $\Psi'_v(G, n/2) \geq \frac{2}{1+\alpha^2}$. Plugging in $\alpha \geq 5\sqrt{2}/8$ for Margulis graphs,

$$\Psi'_v(G_n, n/2) \geq \frac{128}{114} \geq 1.122.$$

Therefore, Margulis graphs have a vertex expansion ratio of 0.122. Using these graphs as G_{SM} in our simulation algorithm, we can tolerate up to $(1 - \frac{1}{2(1.122)}) \geq 0.55n$ failures.

7.4 Impossibility result

We now show an impossibility result about the fault tolerance of consensus in the M&M model. The impossibility depends on the topology of the shared-memory graph G_{SM}. Intuitively, the impossibility indicates that the expander construction is the correct approach: we show that the fault tolerance of the system is related to the minimum cut that separates a large subgraph from the rest of the G_{SM} graph. In graphs with high expansion, the size of such a cut it guaranteed to be large, and thus many failures can be tolerated.

To establish the impossibility, we extend the well-known *partitioning argument* [223] to the M&M model. Basically, if two processes cannot communicate during the execution of an algorithm, then they cannot decide the same value. Thus, if the adversary can partition the system into two disjoint subgraphs A and B, each of size $\geq n - f$, where processes in A do not communicate with processes in B, then agreement cannot hold. This argument works in message-passing models, where the adversary can arbitrarily delay messages on the network, but it breaks in a shared-memory model, in which communication between processes cannot be

delayed without blocking the processes themselves. Thus, in the M&M model, to create such a partition the adversary must get rid of all shared-memory edges of G_{SM} on the cut between A and B.

We now formalize the intuition to arrive at the impossibility. Given a graph $G = (V, E)$, we say that $C = (B, S, T)$ is an *SM-cut* in G if B, S, and T are disjoint subsets of V such that $B \cup S \cup T = V$, and there is a way to partition B into two disjoint subsets B_1 and B_2 such that $(B_1 \cup S, B_2 \cup T)$ is a cut of the graph G, and for every $b_1 \in B_1$, $b_2 \in B_2$, $s \in S$ and $t \in T$, $\{s, t\} \notin E$, $\{b_1, t\} \notin E$, and $\{b_2, s\} \notin E$. Intuitively, B is the set of vertices on the boundary of the cut, and S and T are the remaining vertices on each side.

Theorem 7.4.1. *Consider the M&M model with shared-memory graph G_{SM}, where links are reliable and f processes may crash. Consensus cannot be solved if G_{SM} has an $SM - cut$ (B, S, T) with $|S| \geq n-f$ and $|T| \geq n-f$.*

Note that, in a graph with high expansion, there are no SM-cuts (B, S, T) with $|S| \geq n-f$ and $|T| \geq n-f$. Intuitively, this is because if we want to build an SM-cut and we start with some set S with $|S| \geq n - f$, we must include δS in B_1, and then include $\delta(S \cup B_1)$ in B_2. As these sets expand quickly, we are then left with fewer than $n - f$ vertices to put in T.

Proof. Assume by contradiction that there exists an algorithm A that solves consensus on an M&M network G and is tolerant to $f \geq \min_{C \in SM-cuts(G)} n - |S|$ process failures. Let $C = \arg\min_{C \in SM-cuts(G)} n - |S|$. Consider the following two executions:

EXECUTION E_1: All initial values are 0. All processes in B and in T crash at the beginning and never execute a single step, while all processes in S are correct. This is possible, since $f \geq n - |S| = |B| + |T|$. Then since A solves consensus and is tolerant to f failures, all processes in S must eventually, by some time t, decide 0 and terminate.

EXECUTION E_2: All initial values are 1. All processes in B and in S crash at the beginning and never execute a single step, while all processes in S are correct. This is possible, since by assumption, $|S| \leq |T|$, and therefore $f \geq n - |T| = |B| + |S|$. Then since A solves consensus and is tolerant to f failures, all processes in T must eventually, by some time t', decide 1 and terminate.

We now construct a third execution which violates the specification of consensus.

EXECUTION E_3: The initial values of all processes in S are 0, and the initial values of all processes in T are 1. The initial values of processes in B may be arbitrary. All processes in B crash at the beginning and never execute a single step. Furthermore, all messages from any process in T to any process in S are delayed until after time t, and all messages from any process in S to any process in T are delayed until after time t'. Since all elements of B have crashed, and all shared-memory communication between S and T must pass through B, there cannot be any shared-memory communication. Since this execution is indistinguishable from E_1 to all processes in S until after they decide, and they cannot change their decision, all processes in S decide 0 in E_3. Symmetrically, this execution is indistinguishable from E_2 to processes in T, so they must decide 1. This violates the specification of consensus— a contradiction. □

Note that Theorem 7.4.1 asserts that consensus cannot be solved if there is a SM-cut (B, S, T) in a shared-memory graph where both $|S|$ and $|T|$ are larger than $n - f$. Intuitively, such a SM-cut cannot exist in a shared-memory graph with high expansion. This is because an SM-cut

(B, S, T) can be built in a graph as follows: take any subset S' of the graph. Let $S = S'$, $B_1 = \delta S'$, $B_2 = \delta(S' \cup \delta S')$, and T be the rest of the graph. If necessary, switch S and T such that S is the smaller of the two sets. Since this construction uses the vertex boundaries of sets in the graph, the SM-cut produced would be very small if the sets expand quickly. We formalize this intuition with the following corollary, for which we first define the *2-hop average expansion* of a set.

Definition 7.4.2. *Let $G = (V, E)$. The* 2-hop average expansion ratio *of a set $S \subset V$, denoted $h_S^{(2)}$, is the average of expansion ratios of S and $S \cup \delta S$. That is,*

$$h_S^{(2)} = \frac{1}{2} \cdot \left(\frac{|\delta S|}{|S|} + \frac{|\delta(S \cup \delta S)|}{|S \cup \delta S|} \right)$$

Corollary 7.4.3. *In an M&M network with shared-memory graph $G_{SM} = (V, E)$, consensus cannot be solved if there is a set $S \subset V$ such that $|S| \geq n - f$ and $h_S^{(2)}$ is at most $\sqrt{\frac{f}{|S|}} - 1$.*

Proof. Assume that there exists a set $S \subset V$ such that $|S| \geq n - f$ and $h_S^{(2)} \leq \sqrt{\frac{f}{|S|}} - 1$. We show that in this case, there is an SM-cut in G such that $|S| \geq n - f$ and $|T| \geq n - f$. We do so by constructing the SM-cut (B, S', T) as follows: $S' = S$, $B_1 = \delta S$, $B_2 = \delta(S \cup \delta S)$, and $T = V \setminus (S' \cup B_1 \cup B_2)$. Let $h_1 = \frac{|\delta S|}{|S|}$ and $h_2 = \frac{|\delta(S \cup \delta S)|}{|S \cup \delta S|}$. Then

$$\begin{aligned}
|T| &= n - (|S| + |\delta S| + |\delta(S \cup \delta S)|) \\
&= n - (|S| + h_1|S| + h_2(|S| + h_1|S|) \\
&= n - (|S| + h_1|S| + h_2|S| + h_1 h_2|S|) \\
&\geq n - (1 + 2h_S^{(2)} + (h_S^{(2)})^2)|S| \qquad \text{(by the inequality of means)} \\
&\geq n - \left(1 + 2\left(\sqrt{\frac{f}{|S|}} - 1\right) + \left(\sqrt{\frac{f}{|S|}} - 1\right)^2\right)|S| \\
&= n - \left(1 + \left(\sqrt{\frac{f}{|S|}} - 1\right)\right)^2 \cdot |S| \\
&= n - f
\end{aligned}$$

By Theorem 7.4.1, consensus cannot be solved if there is an SM-cut in G such that $|S| \geq n - f$ and $|T| \geq n - f$. □

Chapter 8

Small RDMA Networks

In the previous chapter, we considered the use of shared memory and message passing together in large networks. In particular, because shared memory is known to be less scalable to large numbers of processes, we assumed that allowing all processes in the system to communicate over shared memory would be impractical. We therefore focused on finding how to improve network guarantees while using only a few well-placed shared memory connections. We showed a trade-off between improved fault tolerance of the overall network and the number of shared memory connections used. The case of a fully-connected shared memory graph, in which all processes were able to communicate with each other over shared memory, was one extreme of the trade-off we considered, and was of limited interest to the discussion of fault tolerance, since the results on such a setting are implied by well-known results in the purely shared memory setting.

In contrast, in this chapter we focus exclusively on the setting in which the network is small enough that allowing all processes to communicate over shared memory is practical. Such small networks in fact crop up very often in the real world, when we only want to optimize communication within a single rack in a data center, or when implementing state machine replication for fault tolerance, where it is common practice to replicate across only a small constant number of machines ($3 - 7$).

In the small network setting, we consider a more detailed version of the M&M model, which reflects more features of RDMA hardware. Recall that RDMA was one of the main technologies that inspired the M&M model in the first place; it provides the traditional send/receive communication primitives, but also allows a process to directly read/write remote memory. Recent work shows that RDMA leads to some new and exciting distributed algorithms [12, 42, 106, 171, 250, 279].

In addition to providing both message-passing and shared memory capabilities, RDMA provides some other features. Namely, RDMA provides a *protection* mechanism to grant and revoke access for reading and writing data. This mechanism is fine grained: an application can divide the memory into different (possibly overlapping) subsets called *memory regions*, and can give different access permissions (read/write/both) to different processes. Furthermore, these protections are *dynamic*: they can be changed by the application over time. However, in RDMA, the remote memories may be subject to failures that cause them to become unresponsive. This behavior differs from traditional shared memory, which is often assumed to be reliable.

In this chapter, we lay the groundwork for a theoretical understanding of these RDMA capabilities by adding them to the M&M model, and we show that they lead to distributed consensus algorithms that are inherently more powerful than before. Similarly to the previous chapter, we focus on small networks in which it is practical to allow all-to-all shared memory communication as enabled by RDMA. Notably, in Chapter 7, where we studied fault tolerance for the consensus problem, the fully-connected setting was of limited interest, since the fault tolerance results were implied by previous results on shared memory systems. However, in this chapter, we consider several aspects that were omitted in Chapter 7.

Namely, we show that RDMA improves on the fundamental trade-off in distributed systems between failure resilience and performance—specifically, we show how a consensus protocol can use RDMA to achieve *both* high resilience and high performance, while traditional algorithms had to choose one or another. As already discussed, traditional shared memory systems provide higher crash-fault tolerance than message passing systems. However, we show that shared memory algorithms are also inherently slower than their message-passing counterparts when solving consensus, as they require at least 2 operations by the same process (4 network delays in our model), even in best-case executions, whereas in message-passing, consensus can be solved in a single network round trip (2 network delays in our model). We formalized these notions of performance in Section 8.2. Furthermore, in this chapter we consider not only crash failures, but also Byzantine faults, whereby processes may arbitrarily deviate from the protocol.

With Byzantine failures, we consider the consensus problem called weak Byzantine agreement, defined by Lamport [192]. We give an algorithm that (a) requires only $n \geq 2f_P + 1$ processes (where f_P is the maximum number of faulty processes) and (b) decides in two delays in the common case. With crash failures, we give the first algorithm for consensus that requires only $n \geq f_P + 1$ processes and decides in two delays in the common case. With both Byzantine or crash failures, our algorithms can also tolerate crashes of memory—only $m \geq 2f_M + 1$ memories are required, where f_M is the maximum number of faulty memories. Furthermore, with crash failures, we improve resilience further, to tolerate crashes of a minority of the combined set of memories and processes.

Our algorithms appear to violate known impossibility results: it is known that with message-passing, Byzantine agreement requires $n \geq 3f_P + 1$, while consensus with crash failures require $n \geq 2f_P + 1$ if the system is partially synchronous [109]. There is no contradiction: our algorithms rely on the power of RDMA, not available in other systems.

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

To see how shared-memory helps resilience, consider the Disk Paxos algorithm [126], which uses shared-memory (disks) but no messages. Disk Paxos requires only $n \geq f_P + 1$ processes, matching the resilience of our algorithm. However, Disk Paxos is not as fast: it takes at least four delays. In fact, we show that no shared-memory consensus algorithm can decide in two delays (Section 8.5).

To see how message-passing helps performance, consider the Fast Paxos algorithm [196], which uses message-passing and no shared-memory. Fast Paxos decides in only two delays in common executions, but it requires $n \geq 2f_P + 1$ processes.

Of course, the challenge is achieving both high resilience and good performance in a single

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

8.1 Related Work

RDMA. Many high-performance systems were recently proposed using RDMA, such as distributed key-value stores [106, 171], communication primitives [106, 173], and shared address spaces across clusters [106]. Kaminsky *et al.* [175] provides guidelines for designing systems using RDMA. RDMA has also been applied to solve consensus [42, 250, 279]. Our model shares similarities with DARE [250] and APUS [279], which modify queue-pair state at run time to prevent or allow access to memory regions, similar to our dynamic permissions. These systems perform better than TCP/IP-based solutions, by exploiting better raw performance of RDMA, without changing the fundamental communication complexity or failure-resilience of the consensus protocol. Similarly, Rüsch *et al.* [258] use RDMA as a replacement for TCP/IP in existing BFT protocols.

Byzantine Fault Tolerance. Lamport, Shostak and Pease [197, 248] show that Byzantine agreement can be solved in synchronous systems iff $n \geq 3f_P + 1$. With unforgeable signatures, Byzantine agreement can be solved iff $n \geq 2f_P + 1$. In asynchronous systems subject to failures, consensus cannot be solved [119]. However, this result is circumvented by making additional assumptions for liveness, such as randomization [51] or partial synchrony [76, 109]. Many Byzantine agreement algorithms focus on safety and implicitly use the additional assumptions for liveness. Even with signatures, asynchronous Byzantine agreement can be solved only if $n \geq 3f_P + 1$ [65].

It is well known that the resilience of Byzantine agreement varies depending on various model assumptions like synchrony, signatures, equivocation, and the exact variant of the problem to be solved. A system that has non-equivocation is one that can prevent a Byzantine process from sending different values to different processes. Table 8.1 summarizes some known results that are relevant to the work in this chapter.

Our Byzantine agreement results share similarities with results for shared memory. Malkhi *et al.* [224] and Alon *et al.* [17] show bounds on the resilience of strong and weak consensus in a model with reliable memory but Byzantine processes. They also provide consensus protocols, using read-write registers enhanced with sticky bits (write-once memory) and access control lists not unlike our permissions. Bessani *et al.* [57] propose an alternative to sticky bits and access control lists through Policy-Enforced Augmented Tuple Spaces. All these works handle Byzantine failures with powerful objects rather than registers. Bouzid *et al.* [63] show that $3f_P+1$ processes are necessary for strong Byzantine agreement with read-write registers.

Work	Synchrony	Signatures	Non-Equiv	Strong Validity	Resiliency
[197]	✓	✓	✗	✓	$2f+1$
[197]	✓	✗	✗	✓	$3f+1$
[17, 224]	✗	✓	✓	✓	$3f+1$
[85]	✗	✓	✗	✗	$3f+1$
[85]	✗	✗	✓	✗	$3f+1$
[85]	✗	✓	✓	✗	$2f+1$
This work	✗	✓	✗ (RDMA)	✗	$2f+1$

Table 8.1: Known fault tolerance results for Byzantine agreement.

Some prior work solves Byzantine agreement with $2f_P+1$ processes using specialized trusted components that Byzantine processes cannot control [82, 83, 93, 94, 176, 277]. Some schemes decide in two delays but require a large trusted component: a coordinator [83], reliable broadcast [94], or message ordering [176]. For us, permission checking in RDMA is a trusted component of sorts, but it is small and readily available.

At a high-level, our improved Byzantine fault tolerance is achieved by preventing equivocation by Byzantine processes, thereby effectively translating each Byzantine failure into a crash failure. Such translations from one type of failure into a less serious one have appeared extensively in the literature [40, 65, 85, 238]. Early work [40, 238] shows how to translate a crash tolerant algorithm into a Byzantine tolerant algorithm in the synchronous setting. Bracha [64] presents a similar translation for the asynchronous setting, in which $n \geq 3f_P + 1$ processes are required to tolerate f_P Byzantine failures. Bracha's translation relies on the definition and implementation of a *reliable broadcast* primitive, very similar to the non-equivocating broadcast in this chapter. However, we show that using the capabilities of RDMA, we can implement it with higher fault tolerance.

Faulty Memory. Afek *et al.* [6] and Jayanti *et al.* [167] study the problem of masking the benign failures of shared memory or objects. We use their ideas of replicating data across memories. Abraham *et al.* [1] considers honest processes but malicious memory.

Common-Case Executions. Many systems and algorithms tolerate adversarial scheduling but optimize for common-case executions without failures, asynchrony, contention, etc (e.g., [61, 103, 108, 180, 191, 196, 225]). None of these match both the resilience and performance of our algorithms. Some algorithms decide in one delay but require $n \geq 5f_P + 1$ for Byzantine failures [270] or $n \geq 3f_P + 1$ for crash failures [66, 103].

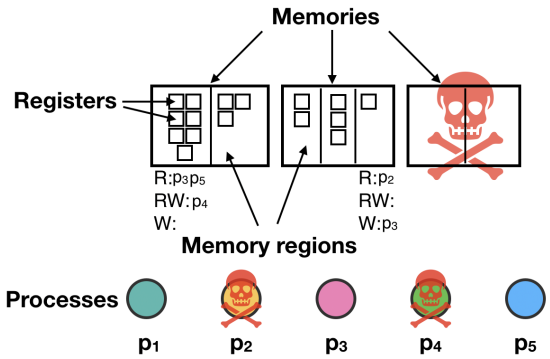


Figure 8.1: Our model with processes and memories, which may both fail. Processes can send messages to each other or access registers in the memories. Registers in a memory are grouped into memory regions that may overlap, but in our algorithms they do not. Each region has a permission indicating what processes can read, write, and read-write the registers in the region (shown for two regions).

8.2 Model

We consider an extension of the message-and-memory (M&M) model; just like is described in Section 6.2, processes may send messages and communicate over shared memory. We assume a fully connected shared memory network. The system has n processes $P = \{p_1, \ldots, p_n\}$ and m (shared) *memories* $M = \{\mu_1, \ldots, \mu_m\}$, which all processes can access. Throughout this chapter, memory refers to the shared memories, not the local state of processes.

Memory Permissions. Each memory consists of a set of *registers*. To control access, an algorithm groups those registers into a set of (possibly overlapping) *memory regions*, and then defines permissions for those memory regions. Formally, a memory region *mr* of a memory μ is a subset of the registers of μ. We often refer to *mr* without specifying the memory μ explicitly. Each memory region *mr* has a *permission*, which consists of three disjoint sets of processes R_{mr}, W_{mr}, RW_{mr} indicating whether each process can read, write, or read-write the registers in the region. We say that p *has read permission on mr* if $p \in R_{mr}$ or $p \in RW_{mr}$; we say that p *has write permission on mr* if $p \in W_{mr}$ or $p \in RW_{mr}$. In the special case when $R_{mr} = P \setminus \{p\}$, $W_{mr} = \emptyset$, $RW_{mr} = \{p\}$, we say that *mr* is a Single-Writer Multi-Reader (SWMR) region—registers in *mr* correspond to the traditional notion of SWMR registers. Note that a register may belong to several regions, and a process may have access to the register on one region but not another—this models the existing RDMA behavior. Intuitively, when reading or writing data, a process specifies the region and the register, and the system uses the region to determine if access is allowed (we make this precise below).

Permission Change. An algorithm indicates an initial permission for each memory region *mr*. Subsequently, the algorithm may wish to change the permission of *mr* during execution. For that, processes can invoke an operation *changePermission*(mr, new_perm), where *new_perm* is a triple (R, W, RW). This operation returns no results and it is intended to modify R_{mr}, W_{mr}, RW_{mr} to R, W, RW. To tolerate Byzantine processes, an algorithm can restrict processes from changing

permissions. For that, the algorithm specifies a function *legalChange*$(p, mr, old\_perm, new\_perm)$ which returns a boolean indicating whether process p can change the permission of *mr* to *new_perm* when the current permissions are *old_perm*. More precisely, when *changePermission* is invoked, the system evaluates *legalChange* to determine whether *changePermission* takes effect or becomes a no-op. When *legalChange* always returns false, we say that the *permissions are static*; otherwise, the *permissions are dynamic*.

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

Executions and Steps. Like in the general model of this thesis in Section 1.5, an execution is as a sequence of process steps. In each step, a process does the following, according to its local state: (1) sends a message or invokes an operation on a memory (read, write, or changePermission), (2) tries to receive a message or a response from an outstanding operation, and (3) changes local state. We require a process to have at most one outstanding operation on each memory.

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

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

Messages and Disks. The model defined above includes two common models as special cases. In the *message-passing* model, there are no memories ($m = 0$), so processes can communicate only by sending messages. In the *disk* model [126], there are no links, so processes can communicate only via memories; moreover, each memory has a single region which always permits all processes to read and write all registers.

Complexity of Algorithms. We are interested in the performance of algorithms in *common-case executions*, when the system is synchronous and there are no failures. In those cases, we measure performance using the notion of *delays*, which extends message-delays to our model.

Under this metric, computations are instantaneous, each message takes one delay, and each memory operation takes two delays. Intuitively, a delay represents the time incurred by the network to transmit a message; a memory operation takes two delays because its hardware implementation requires a round trip. We say that a consensus protocol is k-*deciding* if, in common-case executions, some process decides in k delays.

8.2.1 Reflecting RDMA in Practice

Our model is meant to reflect capabilities of RDMA, while providing a clean abstraction to reason about. We now briefly discuss how features of our model can be implemented using RDMA. Recall that an overview of how RDMA works in practice is presented in Section 6.1.

Memory regions in our model correspond to RDMA memory regions. Using RDMA, a process p can grant permissions to a remote process q by registering memory regions with the appropriate access permissions (read, write, or read/write) and sending the corresponding key to q. p can revoke permissions dynamically by simply de-registering the memory region. Another way for p to change permissions is by changing the queue pair setting. To prevent Byzantine processes from changing permissions illegally, permission changes should be done in the OS kernel, which is less susceptible to corruption. Alternatively, future hardware support similar to SGX could even allow parts of the kernel to be Byzantine without harming our model assumptions.

As highlighted by previous work [250], failures of the CPU, NIC and DRAM can be seen as independent (e.g., arbitrary delays, too many bit errors, failed ECC checks, respectively). For instance, *zombie servers* in which the CPU is blocked but RDMA requests can still be served account for roughly half of all failures [250]. This motivates our choice to treat processes and memory separately in our model. In practice, if a CPU fails permanently, the memory will also become unreachable through RDMA eventually; however, in such cases memory may remain available long enough for ongoing operations to complete. Also, in practical settings it is possible for full-system crashes to occur (e.g., machine restarts), which correspond to a process and a memory failing at the same time—this is allowed by our model.

8.3 Byzantine Agreement

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

The first sub-algorithm, called *Robust Backup*, is developed in two steps. We first implement a primitive called *non-equivocating broadcast*, which prevents Byzantine processes from sending different values to different processes. Then, we use the framework of Clement *et al.* [85] combined with this primitive to convert a message-passing consensus algorithm that tolerates crash failures into a consensus algorithm that tolerates Byzantine failures. This yields Robust Backup.[1]

[1]The attentive reader may wonder why at this point we have not achieved a 2-deciding algorithm already: if we apply Clement *et al.* [85] to a 2-deciding crash-tolerant algorithm (such as Fast Paxos [196]), will the result not be a 2-deciding Byzantine-tolerant algorithm? The answer is no, because Clement *et al.* needs non-equivocated

It uses only static permissions and assumes memories are split into SWMR regions. Therefore, this sub-algorithm works in the traditional shared-memory model with SWMR registers, and it may be of independent interest.

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

Finally, we combine both sub-algorithms using ideas from the Abstract framework of Aublin et al. [31]. More precisely, we start by running Cheap Quorum; if it aborts, we run Robust Backup. There is a subtlety: for this idea to work, Robust Backup must decide on a value v if Cheap Quorum decided v previously. To do that, Robust Backup decides on a *preferred value* if at least $f+1$ processes have this value as input. To do so, we use the classic crash-tolerant Paxos algorithm (run under the Robust Backup algorithm to ensure Byzantine tolerance) but with an initial set-up phase that ensures this safe decision. We call the protocol *Preferential Paxos*.

We develop Robust Backup using the construction by Clement *et al.* [85], which we now explain. Clement *et al.* show how to transform a message-passing algorithm $\mathcal{A}$ that tolerates f_P crash failures into a message-passing algorithm that tolerates f_P Byzantine failures in a system where $n \geq 2f_P + 1$ processes, assuming unforgeable signatures and a non-equivocation mechanism. They do so by implementing trusted message-passing primitives, *T-send* and *T-receive*, using non-equivocation and signature verification on every message. Processes include their full history with each message, and then verify locally whether a received message is consistent with the protocol. This restricts Byzantine behavior to crash failures.

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

8.3.1 Non-Equivocating Broadcast

Consider a shared-memory system. We present a way to prevent equivocation through a solution to the following broadcast problem, which we call *non-equivocating broadcast*. This notion is similar to reliable broadcast, but makes use of a sequence number, k.

Definition 8.3.1. Non-equivocating broadcast *is defined in terms of two primitives, broadcast*(k, m) *and deliver*(k, m, q). *When a process* p *invokes broadcast*(k, m) *we say that* p broadcasts (k, m). *When a process* p *invokes deliver*(k, m, q) *we say that* p delivers (k, m) from q. *Each correct process* p *must invoke broadcast*$(k, *)$ *with* k *one higher than* p*'s previous invocation (and first invocation with* k=1*). The following holds:*

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

broadcast, which incurs at least 6 delays.

4. *If a correct process delivers (k, m) from p, then all correct processes eventually deliver (k, m') from p for some m'.*

Algorithm 8.2 shows how to implement non-equivocating broadcast that is tolerant to a minority of Byzantine failures in shared-memory using SWMR registers.

To broadcast its k-th message m, p simply signs (k, m) and writes it in slot $Value[p, k, p]$ of its memory[2].

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

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

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

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

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

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

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

This three-stage validation process ensures the following crucial property: no two valid L2

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

Algorithm 8.2: Non-Equivocating Broadcast

```
1  SWMR Value[n,M,n]; //Initially, Value[j,k,i]=⊥.
2  SWMR L1Proof[n,M,n]; //Initially, L1Proof[j,k,i]=⊥.
3  SWMR L2Proof[n,M,n]; //Initially, L2Proof[j,k,i]=⊥.

5  Code for process p
6   last[n]; //last index delivered from each process. Initially, last[q]=0
7   state[n]; //∈{WaitForSender,WaitForL1Proof,WaitForL2Proof}. Initially,
        ↪ state[q]=WaitForSender

9   void broadcast (k,m){
10    Write(Value[p,k,p], sign((k,m)));  }

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

16  void try_deliver(q) {
17    k = last[q];
18    val = checkL2Proof(q,k);
19    if (val != null) {
20      deliver(k, val, q);
21      last[q] += 1;
22      state = WaitForSender;
23      return; }
24    my_val = Read(Value[p,k,q]);

26    if (state == WaitForSender) {
27      my_val = Read(Value[q,k,q]);
28      if (my_val==⊥ || !sValid(q, val) || key!=k) { return; }
29      Write(Value[p,k,q],sign(my_val));
30      state = WaitForL1Proof; }

32    if (state == WaitForL1Proof) {
33      checkedVals = ∅;
34      for i ∈Π{
35        v = Read(Value[i,k,q]);
36        if (validateValue(v,my_val,k,q)) {checkedVals.add((i,v));} }

38      if (size(checkedVals) > n/2) {
39        l1prf = sign(checkedVals);
40        Write(L1Proof[p,k,q], l1prf);
41        state = WaitForL2Proof; } }

43    if (state == WaitForL2Proof){
44      checkedL1Prfs = ∅;
45        for i in Π{
46          prf = Read(L1Proof[i,k,q]);
47          if (validateL1Prf(prf,my_val,k,q)) {checkedL1Prfs.add((i,prf));}}

49      if (size(checkedL1Prfs) > n/2) {
50        l2prf = sign(checkedL1Prfs);
51        Write(L2Proof[p,k,q], l2prf); } } }
```

Algorithm 8.3: Helper functions for Non-Equivocating Broadcast algorithm

```
52 value checkL2proof(q,k) {
53   for i∈ Π {
54     proof = Read(L2Proof[i,k,q]);
55     if (proof != ⊥){
56       val = first value in proof; }
57     if (proof != ⊥ && validateL2Prf(proof,val,k,q)) {
58       Write(L2Proof[p,k,q],proof);
59       return val; }   }
60   return null;    }

62 bool validateValue(v, val, k, q){
63   if (v == val && sValid(q,v) && key == k){
64     return true; }
65   return false; }

67 bool validateL1Prf(proof, val, k, q){
68   if (size(proof) > n/2) {
69     for each (i,(v, s)) ∈ proof {
70       if (!validateValue(v,val,k,q) || !sValid(i, (v,s)){
71         return false; }   } }
72   return true;  }

74 bool validateL1Prf(proof, val, k, q){
75   if (size(proof) > n/2 && ∀
76     for each (i,(l1prf,s)) ∈ proof {
77       if (!validateL1Proof(l1prf,val,k,q) || !sValid(i, (l1prf, s))){
78         return false;}  }   }
79   return true;  }
```

proofs can support different values. Intuitively, this property is achieved because for both L1 and L2 proofs, at least $f + 1$ values of the previous stage must be copied, meaning that at least one correct process was involved in the quorum needed to construct each proof. Because correct processes read the slots of *all* others at each stage before constructing the next proof, and because they never overwrite or delete values that they already wrote, it is guaranteed that no two correct processes will create valid L1 proofs for different values, since one must see the Value slot of the other. Thus, two no process, Byzantine or otherwise, can construct valid L2 proofs for different values.

Notably, a weaker version of non-equivocating broadcast, which does not require Property 4, can be solved with just the first stage of Algorithm 8.2, without the L1 and L2 proofs. The purpose of those proofs is to ensure the 4th property holds; that is, to enable all correct processes to deliver a value once some correct process delivered.

Correctness of Non-Equivocating Broadcast. We now prove the following key lemma:

Lemma 8.3.2. *Non-equivocating broadcast can be solved in shared-memory with SWMR regular registers with $n \geq 2f + 1$ processes.*

We begin the proof with a couple of useful observations.

Observation 8.3.3. *In Algorithm 8.2, if p is a correct process, then no slot that belongs to p is written to more than once.*

Proof. Since p is correct, p never writes on any slot more than once. Furthermore, since all slots are single-writer registers, no other process can write on these slots. □

Observation 8.3.4. *In Algorithm 8.2, correct processes invoke and return from try_deliver(q) infinitely often, for all $q \in \Pi$.*

Proof. The try_deliver() function does not contain any blocking steps, loops or goto statements. Thus, if a correct process invokes try_deliver(), it will eventually return. Therefore, for a fixed q the infinite loop at line 14 will invoke and return try_deliver(q) infinitely often. Since the parallel for loop at line 12 performs the infinite loop in parallel for each $q \in \Pi$, the Observation holds. □

Proof of Lemma 8.3.2. We prove the lemma by showing that Algorithm 8.2 correctly implements non-equivocating broadcast. That is, we need to show that Algorithm 8.2 satisfies the four properties of non-equivocating broadcast.

Property 1. Let p be a correct process that broadcasts (k, m). We show that all correct processes eventually deliver (k, m) from p. Assume by contradiction that there exists some correct process q which does not deliver (k, m). Furthermore, assume without loss of generality that k is the smallest key for which Property 1 is broken. That is, all correct processes must eventually deliver all messages (k', m') from p, for $k' < k$. Thus, all correct processes must eventually increment $last[p]$ to k.

We consider two cases, depending on whether or not some process eventually writes an L2 proof for some (k, m') message from p in its L2Proof slot.

First consider the case where no process ever writes an L2 proof of any value (k, m') from p. Since p is correct, upon broadcasting (k, m), p must sign and write (k, m) into Value[p,k,p]

at line 10. By Observation 8.3.3, (k, m) will remain in that slot forever. Because of this, and because there is no L2 proof, all correct processes, after reaching $last[p] = k$, will eventually read (k, m) in line 27, write it into their own Value slot in line 29 and change their state to WaitForL1Proof.

Furthermore, since p is correct and we assume signatures are unforgeable, no process q can write any other valid value $(k', m') \neq (k, m)$ into its $Values[q, k, p]$ slot. Thus, eventually each correct process q will add at least $f+1$ copies of (k, m) to its checkedVals, write an L1proof consisting of these values into L1Proof[q,k,p] in line 40, and change their state to WaitForL2Proof.

Therefore, all correct processes will eventually read at least $f + 1$ valid L1 Proofs for (k, m) in line 46 and construct and write valid L2 proofs for (k, m). This contradicts the assumption that no L2 proof ever gets written.

In the case where there is some L2 proof, by the argument above, the only value it can prove is (k, m). Therefore, all correct processes will see at least one valid L2 proof at deliver. This contradicts our assumption that q is correct but does not deliver (k, m) from p.

Property 2. We now prove the second property of non-equivocating broadcast. Let p and p' be any two correct processes, and q be some process, such that p delivers (k, m) from q and p' delivers (k, m') from q. Assume by contradiction that $m \neq m'$.

Since p and p' are correct, they must have seen valid L2 proofs at line 54 before delivering (k, m) and (k, m') respectively. Let $\mathcal{P}$ and $\mathcal{P}'$ be those valid proofs for (k, m) and (k, m') respectively. $\mathcal{P}$ (resp. $\mathcal{P}'$) consists of at least $f + 1$ valid L1 proofs; therefore, at least one of those proofs was created by some correct process r (resp. r'). Since r (resp. r') is correct, it must have written (k, m) (resp. (k, m')) to its Values slot in line 29. Note that after copying a value to their slot, in the WaitForL1Proof state, correct processes read *all* Value slots line 35. Thus, both r and r' read all Value slots before compiling their L1 proof for (k, m) (resp. (k, m')).

Assume without loss of generality that r wrote (k, m) before r' wrote (k, m'); by Observation 8.3.3, it must then be the case that r' later saw both (k, m) and (k, m') when it read all Values slots (line 35). Since r' is correct, it cannot have then compiled an L1 proof for (k, m') (the check at line 38 failed). We have reached a contradiction.

Property 3. We show that if a correct process p delivers (k, m) from a correct process p', then p' broadcast (k, m). Correct processes only deliver values for which a valid L2 proof exists (lines 54—20). Therefore, p must have seen a valid L2 proof $\mathcal{P}$ for (k, m). $\mathcal{P}$ consists of at least $f + 1$ L1 proofs for (k, m) and each L1 proof consists of at least $f + 1$ matching copies of (k, m), signed by p'. Since p' is correct and we assume signatures are unforgeable, p' must have broadcast (k, m) (otherwise p' would not have attached its signature to (k, m)).

Property 4. Let p be a correct process such that p delivers (k, m) from q. We show that all correct process must deliver (k, m') from q, for some m'.

By construction of the algorithm, if p delivers (k, m) from q, then for all $i < k$ there exists m_i such that p delivered (i, m_i) from q before delivering (k, m) (this is because p can only deliver (k, m) if $last[q] = k$ and $last[q]$ is only incremented to k after p delivers $(k - 1, m_{k-1})$).

Assume by contradiction that there exists some correct process r which does not deliver (k, m') from q, for any m'. Further assume without loss of generality that k is the smallest key for which r does not deliver any message from q. Thus, r must have delivered (i, m'_i) from q for all $i < k$; thus, r must have incremented $last[q]$ to k. Since r never delivers any message from q for key k, r's $last[q]$ will never increase past k.

Algorithm 8.4: Validate Operation for Non-Equivocating Broadcast

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

Since p delivers (k, m) from q, then p must have written a valid L2 proof $\mathcal{P}$ of (k, m) in its L2Proof slot in line 51 or 58. By Observation 8.3.3, $\mathcal{P}$ will remain in p's L2Proof[p,k,q] slot forever. Thus, at least one of the slots $L2Proof[\cdot, k, q]$ will forever contain a valid L2 proof. Since r's $last[q]$ eventually reaches k and never increases past k, r will eventually (by Observation 8.3.4) see a valid L2 proof in line 54 and deliver a message for key k from q. We have reached a contradiction. □

Applying Clement et al's Construction. Clement et al. show that given unforgeable transferable signatures and non-equivocation, one can reduce Byzantine failures to crash failures in message passing systems [85]. They define non-equivocation as a predicate $valid_p$ for each process p, which takes a sequence number and a value and evaluates to true for just one value per sequence number. All processes must be able to call the same $valid_p$ predicate, which always terminates every time it is called.

We now show how to use non-equivocating broadcast to implement messages with transferable signatures and non-equivocation as defined by Clement et al. [85]. Note that our non-equivocating broadcast mechanism already involves the use of transferable signatures, so to send and receive signed messages, one can simply use broadcast and deliver those messages. However, simply using broadcast and deliver is not quite enough to satisfy the requirements of the $valid_p$ predicate of Clement et al. The problem occurs when trying to validate nested messages recursively.

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

To solve this issue, we introduce a `validate` operation that can be used along with broadcast and deliver to validate the correctness of a given message. The `validate` operation is very simple: it takes in a process id, a sequence number, and a message value m, and simply runs the `checkL2proof` helper function. If the function returns a proof supporting m, `validate` returns true. Otherwise it returns false. The pseudocode is shown in Algorithm 8.4.

In this way, Algorithms 8.2 and 8.4 together provide signed messages and a non-equivocation primitive. Thus, combined with the construction of Clement et al. [85], we immediately get the

Algorithm 8.5: T-send and T-receive (due to Clement et al. [85]) with non-equivocating broadcast.

```
1  Local variables for each process:
2  int k = 0
3  history H = []
4  T-send(m) {
5    k++
6    broadcast(k,(m,H))
7    Append "sent(k,(m,H))" to H }

9  Upon deliver(k,(m,H),p):
10   Check whether all messages in H are properly signed, and whether they
       ↪ correspond to a correct history of the algorithm
11   if so,
12     T-receive(m,p)
13     add "received(sign(k,(m,H), p))" to H
```

following result.

Theorem 8.3.5. *There exists an algorithm for weak Byzantine agreement in a shared-memory system with SWMR regular registers, signatures, and up to f_P process crashes where $n \geq 2f_P + 1$.*

In particular, we can implement weak Byzantine agreement by taking any correct consensus algorithm $\mathcal{A}$ for the classic crash-only message passing model, and replacing all its sends and receives by non-equivocating broadcast and deliver (respectively) that also attach a process's entire execution history to each message. We call this method of communication *trusted sends and receives*, or simply T-send and T-receive primitives. Clement et al. [85] show that implementing such T-send and T-receive primitives with non-equivocation and signatures yields a Byzantine-tolerant replacement for classic sends and receives.

Algorithm 8.5 shows how to use non-equivocating broadcast to implement T-send and T-receive. See Clement et al. [85] for more details.

Non-Equivocation in Our Model. To convert the above algorithm to our model, where memory may fail, we use the ideas in [6, 29, 167] to implement failure-free SWMR regular registers from the fail-prone memory, and then run weak Byzantine agreement using those regular registers. To implement an SWMR register, a process writes or reads all memories, and waits for a majority to respond. When reading, if p sees exactly one distinct non-$\perp$ value v across the memories, it returns v; otherwise, it returns $\perp$.

Definition 8.3.6. *Let $\mathcal{A}$ be a message-passing algorithm.* Robust Backup($\mathcal{A}$) *is the algorithm $\mathcal{A}$ in which all $send$ and $receive$ operations are replaced by T-send and T-receive operations (respectively) implemented with non-equivocating broadcast.*

Thus we get the following lemma, from the result of Clement et al. [85], Lemma 8.3.2, and the above handling of memory failures.

Lemma 8.3.7. *If $\mathcal{A}$ is a consensus algorithm that is tolerant to f process crash failures, then Robust Backup($\mathcal{A}$) is a weak Byzantine agreement algorithm that is tolerant to up to f_P Byzantine processes and f_M memory crashes, where $n \geq 2f_P + 1$ and $m \geq 2f_M + 1$ in the message-and-*

Algorithm 8.6: Cheap Quorum normal operation—code for process p

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

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

memory model.

The following theorem is an immediate corrolary of the lemma.

Theorem 8.3.8. *There exists an algorithm for Weak Byzantine Agreement in a message-and-memory model with up to f_P Byzantine processes and f_M memory crashes, where $n \geq 2f_P + 1$ and $m \geq 2f_M + 1$.*

8.3.2 The Cheap Quorum Sub-Algorithm

We now give an algorithm that decides in two delays in common executions in which the system is synchronous and there are no failures. It requires only one signature for a fast decision, whereas the best prior algorithm requires $6f_P + 2$ signatures and $n \geq 3f_P + 1$ [31]. Our algorithm, called Cheap Quorum, is not in itself a complete consensus algorithm; it may abort in some executions. If Cheap Quorum aborts, it outputs an *abort value*, which is used to initialize the Robust Backup so that their composition preserves weak Byzantine agreement. This composition is inspired by the Abstract framework of Aublin *et al.* [31].

The algorithm has a special process ℓ, say $\ell = p_1$, which serves both as a *leader* and a *follower*. Other processes act only as *followers*. The memory is partitioned into $n + 1$ regions denoted *Region*$[p]$ for each $p \in \Pi$, plus an extra one for p_1, *Region*$[\ell]$ in which it proposes a value. Initially, *Region*$[p]$ is a regular SWMR region where p is the writer. Unlike in Algorithm 8.2, some of the permissions are dynamic; processes may remove p_1's write permission to *Region*$[\ell]$

Algorithm 8.7: Cheap Quorum panic mode—code for process p

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

(i.e., the *legalChange* function returns false to any permission change requests, except for ones revoking p_1's permission to write on *Region*$[\ell]$).

Processes initially execute under a *normal* mode in common-case executions, but may switch to *panic* mode if they intend to abort, as in [31]. The pseudo-code of the normal mode is in Algorithm 8.6. *Region*$[p]$ contains three registers *Value*$[p]$, *Panic*$[p]$, *Proof*$[p]$ initially set to $\perp$, *false*, $\perp$. To propose v, the leader p_1 signs v and writes it to *Value*$[\ell]$. If the write is successful (it may fail because its write permission was removed), then p_1 decides v; otherwise p_1 calls *Panic_mode*$()$. Note that all processes, including p_1, continue their execution after deciding. However, p_1 never decides again if it decided as the leader. A follower q checks if p_1 wrote to *Value*$[\ell]$ and, if so, whether the value is properly signed. If so, q signs v, writes it to *Value*$[q]$, and waits for other processes to write the same value to *Value*$[*]$. If q sees $2f + 1$ copies of v signed by different processes, q assembles these copies in a *unanimity proof*, which it signs and writes to *Proof*$[q]$. q then waits for $2f + 1$ unanimity proofs for v to appear in *Proof*$[*]$, and checks that they are valid, in which case q decides v. This waiting continues until a timeout expires[3], at which time q calls *Panic_mode*$()$. In *Panic_mode*$()$, a process p sets *Panic*$[p]$ to *true* to tell other processes it is panicking; other processes periodically check to see if they should panic too. p then removes write permission from *Region*$[\ell]$, and decides on a value to abort: either *Value*$[p]$ if it is non-$\perp$, *Value*$[\ell]$ if it is non-$\perp$, or p's input value. If p has a unanimity proof in *Proof*$[p]$, it adds it to the abort value.

The above construction assumes a fail-free memory with regular registers, but we can extend it to tolerate memory failures using the approach discussed for making non-equivocating broadcast work in our model, noting that each register has a single writer process.

Correctness of Cheap Quorum. We prove that Cheap Quorum satisfies certain useful properties that will help us show that it composes with Preferential Paxos to form a correct weak Byzantine agreement protocol. For the proofs, we first formalize some terminology. We say that a process *proposed* a value v by time t if it successfully executes line 4; that is, p receives the response ack in line 4 by t. When a process aborts, note that it outputs a tuple. We say that the first element of its tuple is its *abort value*, and the second is its *abort proof*. We sometimes say

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

that a process p aborts with value v and proof pr, meaning that p outputs (v, pr) in its abort. Furthermore, the value in a process p's Proof region is called a *correct unanimity proof* if it contains n copies of the same value, each correctly signed by a different process.

Observation 8.3.9. *In Cheap Quorum, no value written by a correct process is ever overwritten.*

Proof. By inspecting the code, we can see that the correct behavior is for processes to never overwrite any values. Furthermore, since all regions are initially single-writer, and the legalChange function never allows another process to acquire write permission on a region that they cannot write to initially, no other process can overwrite these values. □

Lemma 8.3.10 (Cheap Quorum Validity)**.** *In Cheap Quorum, if there are no faulty processes and some process decides v, then v is the input of some process.*

Proof. If $p = p_1$, the lemma is trivially true, because p_1 can only decide on its input value. If $p \neq p_1$, p can only decide on a value v if it read that value from the leader's region. Since only the leader can write to its region, it follows that p can only decide on a value that was proposed by the leader (p_1). □

Lemma 8.3.11 (Cheap Quorum Termination)**.** *If a correct process p proposes some value, every correct process q will decide a value or abort.*

Proof. Clearly, if $q = p_1$ proposes a value, then q decides. Now let $q \neq p_1$ be a correct follower and assume p_1 is a correct leader that proposes v. Since p_1 proposed v, p_1 was able to write v in the leader region, where v remains forever by Observation 8.3.9. Clearly, if q eventually enters panic mode, then it eventually aborts; there is no waiting done in panic mode. If q never enters panic mode, then q eventually sees v on the leader region and eventually finds $2f + 1$ copies of v on the regions of other followers (otherwise q would enter panic mode). Thus q eventually decides v. □

Lemma 8.3.12 (Cheap Quorum Progress)**.** *If the system is synchronous and all processes are correct, then no correct process aborts in Cheap Quorum.*

Proof. Assume the contrary: there exists an execution in which the system is synchronous and all processes are correct, yet some process aborts. Processes can only abort after entering panic mode, so let t be the first time when a process enters panic mode and let p be that process. Since p cannot have seen any other process declare panic, p must have either timed out at line 12 or 22, or its checks failed on line 13. However, since the entire system is synchronous and p is correct, p could not have panicked because of a time-out at line 12. So, p_1 must have written its value v, correctly signed, to p_1's region at a time $t' < t$. Therefore, p also could not have panicked by failing its checks on line 13. Finally, since all processes are correct and the system is synchronous, all processes must have seen p_1's value and copied it to their slot. Thus, p must have seen these values and decided on v at line 20, contradicting the assumption that p entered panic mode. □

Lemma 8.3.13 (Cheap Quorum Decision Agreement)**.** *Let p and q be correct processes. If p decides v_1 while q decides v_2, then $v_1 = v_2$.*

Proof. Assume the property does not hold: p decided some value v_1 and q decided some different value v_2. Since p decided v_1, then p must have seen a copy of v_1 at $2f_P + 1$ replicas, including q. But then q cannot have decided v_2, because by Observation 8.3.9, v_1 never gets overwritten from q's region, and by the code, q only can decide a value written in its region. □

Lemma 8.3.14 (Cheap Quorum Abort Agreement)**.** *Let p and q be correct processes (possibly identical). If p decides v in Cheap Quorum while q aborts from Cheap Quorum, then v will be q's abort value. Furthermore, if p is a follower, q's abort proof is a correct unanimity proof.*

Proof. If $p = q$, the property follows immediately, because of lines 4 through 6 of panic mode. If $p \neq q$, we consider two cases:

- If p is a follower, then for p to decide, all processes, and in particular, q, must have replicated both v and a correct proof of unanimity before p decided. Therefore, by Observation 8.3.9, v and the unanimity proof are still there when q executes the panic code in lines 4 through 6. Therefore q will abort with v as its value and a correct unanimity proof as its abort proof.
- If p is the leader, then first note that since p is correct, by Observation 8.3.9 v remains the value written in the leader's Value region. There are two cases. If q has replicated a value into its Value region, then it must have read it from $Value[p_1]$, and therefore it must be v. Again by Observation 8.3.9, v must still be the value written in q's Value region when q executes the panic code. Therefore q aborts with value v. Otherwise, if q has not replicated a value, then q's Value region must be empty at the time of the panic, since the legalChange function disallows other processes from writing on that region. Therefore q reads v from $Value[p_1]$ and aborts with v. □

Lemma 8.3.15. *Cheap Quorum is 2-deciding.*

Proof. Consider an execution in which every process is correct and the system is synchronous. Then no process will enter panic mode (by Lemma 8.3.12) and thus p_1 will not have its permission revoked. p_1 will therefore be able to write its input value to p_1's region and decide after this single write (2 delays). □

8.3.3 Putting it Together: the Cheap & Robust Algorithm

The final algorithm, called Cheap & Robust, combines Cheap Quorum (Section 8.3.2) and Robust Backup (Section 8.3.1), as we now explain. Recall that Robust Backup is parameterized by a message-passing consensus algorithm $\mathcal{A}$ that tolerates crash-failures. $\mathcal{A}$ can be any such algorithm (e.g., Paxos).

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

For this purpose, we propose a simple wrapper for Robust Backup, called *Preferential Paxos*. Preferential Paxos first runs a set-up phase, in which processes may adopt new values, and then runs Robust Backup with the new values. More specifically, there are some *preferred* input values

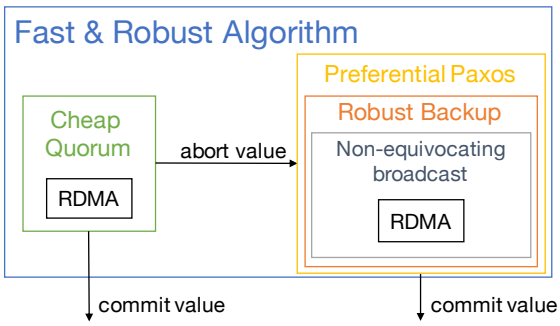


Figure 8.8: Interactions of the components of the Cheap & Robust Algorithm.

$v_1 \dots v_k$, ordered by priority. We guarantee that every process adopts one of the top $f+1$ priority inputs. In particular, this means that if a majority of processes get the highest priority value, v_1, as input, then v_1 is guaranteed to be the decision value. The set-up phase is simple; all processes send each other their input values. Each process p waits to receive $n - f$ such messages, and adopts the value with the highest priority that it sees. This is the value that p uses as its input to Paxos.

The following is the pseudocode of Preferential Paxos. Recall that *T-send* and *T-receive* are the trusted message passing primitives that are implemented in [85] using non-equivocating broadcast and signatures.

Algorithm 8.9: Preferential Paxos—code for process p

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

Correctness of Preferential Paxos.

Lemma 8.3.16 (Preferential Paxos Priority Decision)**.** *Preferential Paxos implements weak Byzantine agreement with $n \geq 2f_P + 1$ processes. Furthermore, let $v_1, \dots, v_n$ be the input values of an instance C of Preferential Paxos, ordered by priority. The decision value of correct processes is always one of $v_1, \dots, v_{f+1}$.*

Proof. By Lemma 8.3.7, Robust Backup(Paxos) solves weak Byzantine agreement with $n \geq 2f_P + 1$ processes. Note that before calling Robust Backup(Paxos), each process may change its input, but only to the input of another process. Thus, by the correctness and fault tolerance of Paxos, Preferential Paxos clearly solves weak Byzantine agreement with $n \geq 2f_P + 1$ processes. Thus we only need to show that Preferential Paxos satisfies the priority decision property with $2f_P + 1$ processes that may only fail by crashing.

Since Robust Backup(Paxos) satisfies validity, if all processes call Robust Backup(Paxos) in line 5 with a value v that is one of the $f_P + 1$ top priority values (that is, $v \in \{v_1, \dots, v_{f_P+1}\}$),

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

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

Definition 8.3.17 (Input Priorities for Preferential Paxos)**.** *The input values for Preferential Paxos as it is used in Cheap & Robust are split into three sets (here, p_1 is the leader of Cheap Quorum):*

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

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

Figure 8.8 shows how the various algorithms presented in this section come together to form the Cheap & Robust algorithm.

Correctness of the Cheap & Robust. We now prove the following key composition property that shows that the composition of Cheap Quorum and Preferential Paxos is safe.

Lemma 8.3.18 (Composition Lemma)**.** *If some correct process decides a value v in Cheap Quorum before an abort, then v is the only value that can be decided in Preferential Paxos with priorities as defined in Definition 8.3.17.*

Proof. To prove this lemma, we mainly rely on two properties: the Cheap Quorum Abort Agreement (Lemma 8.3.14) and Preferential Paxos Priority Decision (Lemma 8.3.16). We consider two cases.

Case 1. Some correct follower process $p \neq p_1$ decided v in Cheap Quorum. Then note that by Lemma 8.3.14, all correct processes aborted with value v and a correct unanimity proof. Since $n \geq 2f+1$, there are at least $f+1$ correct processes. Note that by the way we assign priorities to inputs of Preferential Paxos in the composition of the two algorithms, all correct processes have inputs with the highest priority. Therefore, by Lemma 8.3.16, the only decision value possible in Preferential Paxos is v. Furthermore, note that by Lemma 8.3.13, if any other correct process decided in Cheap Quorum, that process's decision value was also v.

Case 2. Only the leader, p_1, decides in Cheap Quorum, and p_1 is correct. Then by Lemma 8.3.14, all correct processes aborted with value v. Since p_1 is correct, v is signed by p_1. It is possible that some of the processes also had a correct unanimity proof as their abort proof. However, note that in this scenario, all correct processes (at least $f + 1$ processes) had inputs with either the highest or second highest priorities, all with the same abort value. Therefore, by Lemma 8.3.16,

the decision value must have been the value of one of these inputs. Since all these inputs had the same value v, v must be the decision value of Preferential Paxos. □

Theorem 8.3.19 (End-to-end Validity)**.** *In the Cheap & Robust algorithm, if there are no faulty processes and some process decides v, then v is the input of some process.*

Proof. Note that by Lemmas 8.3.10 and 8.3.16, this holds for each of the algorithms individually. Furthermore, recall that the abort values of Cheap Quorum become the input values of Preferential Paxos, and the set-up phase does not invent new values. Therefore, we just have to show that if Cheap Quorum aborts, then all abort values are inputs of some process. Note that by the code in panic mode, if Cheap Quorum aborts, a process p can output an abort value from one of three sources: its own Value region, the leader's Value region, or its own input value. Clearly, if its abort value is its input, then we are done. Furthermore note that a correct leader only writes its input in the Value region, and correct followers only write a copy of the leader's Value region in their own region. Since there are no faults, this means that only the input of the leader may be written in any Value region, and therefore all processes always abort with some processes input as their abort value. □

Theorem 8.3.20 (End-to-end Agreement)**.** *In the Cheap & Robust algorithm, if p and q are correct processes such that p decides v_1 and q decides v_2, then $v_1 = v_2$.*

Proof. First note that by Lemmas 8.3.13 and 8.3.16, each of the algorithms satisfy this individually. Thus Lemma 8.3.18 implies the theorem. □

Theorem 8.3.21 (End-to-end Termination)**.** *In Cheap & Robust algorithm, if some correct process is eventually the sole leader forever, then every correct process eventually decides.*

Proof. Assume towards a contradiction that some correct process p is eventually the sole leader forever, and let t be the time when p last becomes leader. Now consider some process q that has not decided before t. We consider several cases:

1. If q is executing Preferential Paxos at time t, then q will eventually decide, by termination of Preferential Paxos (Lemma 8.3.16).
2. If q is executing Cheap Quorum at time t, we distinguish two sub-cases:
   (a) p is also executing as the leader of Cheap Quorum at time t. Then p will eventually propose a value, so q will either decide in Cheap Quorum or abort from Cheap Quorum (by Lemma 8.3.11) and decide in Preferential Paxos by Lemma 8.3.16.
   (b) p is executing in Preferential Paxos. Then p must have panicked and aborted from Cheap Quorum. Thus, q will also abort from Cheap Quorum and decide in Preferential Paxos by Lemma 8.3.16. □

Note that to strengthen 8.3.21 to general termination as stated in our model, we require the additional standard assumption [193] that some correct process p is eventually the sole leader forever. In practice, however, p does not need to be the sole leader forever, but rather *long enough* so that all correct processes decide.

Theorem 8.3.22. *There exists a 2-deciding algorithm for Weak Byzantine Agreement in a message-and-memory model with up to f_P Byzantine processes and f_M memory crashes, where $n \geq 2f_P + 1$ and $m \geq 2f_M + 1$.*

8.3.4 Impossibility of Strong Agreement

In this chapter and in this section so far, the validity requirement for Byzantine agreement is *weak*; that is, if there is even one Byzantine fault, the processes may agree on an arbitrary value. Another notion of validity, known as *strong validity*, exists in the literature [197]:

Definition 8.3.23 (Strong Validity)**.** *If some process decided the value v, then v was the input of some correct process.*

The Strong Byzantine Agreement problem requires an algorithm to satisfy Agreement and Termination as defined in Section 6.2, and Strong Validity. In this subsection, we show that it is impossible to solve Strong Byzantine Agreement in an RDMA system with $n \leq 3f$. Thus, our solution for the Weak Byzantine Agreement problem cannot be strengthened to Strong Agreement without sacrificing the fault tolerance that RDMA provides.

Theorem 8.3.24. *Strong Byzantine Agreement cannot be solved in a permissions M&M network where $n \leq 3f$.*

Proof. Assume by contradiction that there exists an algorithm, $\mathcal{A}$, that solves strong Byzantine agreement with $n \leq 3f$ in this setting. We partition the set of processes into three subsets, A, B, and C, such that the size of each set $\leq f$. Consider the following scenarios.

Scenario 1. All processes in A and B are correct and have input 1. Processes in C are faulty, but act as if they are correct and have input 0. Processes in B sleep at the beginning of the execution, not taking any steps. We run $\mathcal{A}$ until processes in A decide some value v. Then, processes in B wake up, and run $\mathcal{A}$. Since $\mathcal{A}$ is a correct strong Byzantine agreement algorithm, processes in B eventually also decide the value v. Furthermore, since all correct processes (those in the sets A and B) have the same input value 1, $v = 1$.

Scenario 2. All processes in C and B are correct and have input 0. Processes in A are faulty, but act as if they are correct and have input 1. Processes in B sleep at the beginning of the execution, not taking any steps. We run $\mathcal{A}$ until processes in C decide some value v'. Then, processes in B wake up, and run $\mathcal{A}$. Since $\mathcal{A}$ is a correct strong Byzantine agreement algorithm, processes in B eventually also decide the value v'. Furthermore, since all correct processes (those in the sets C and B) have the same input value 0, $v' = 0$.

Scenario 3. All processes in A and C are correct. Processes in A have input value 1, and processes in C have input value 0. Processes in B are faulty and never take any steps. Since $\mathcal{A}$ is a correct strong Byzantine agreement algorithm, processes in A and C eventually decide the same value v''.

Note that Scenario 1 and Scenario 3 are indistinguishable to processes in A. Thus, since in Scenario 1 the decision value is 1, there must be an execution of Scenario 3 in which the decision value is 1. Furthermore, Scenario 2 and Scenario 3 are indistinguishable to processes in C, and thus there must be an execution of Scenario 3 in which the decision value is 0.

Consider an execution of Scenario 3 that decides 0. Run this execution in Scenario 1. Since Scenario 1 is indistinguishable to processes in A from Scenario 3, processes in A decide 0 in

this execution. However, this violates strong validity, since in Scenario 1, all processes had input 1. □

We note that the proof in fact applies more generally than to just the RDMA model. A similar proof, that considers shared memory objects, appears in [224].

8.4 Crash-Tolerant Consensus

We now restrict ourselves to crash failures of processes and memories. Clearly, we can use the algorithms of Section 8.3 in this setting, to obtain a 2-deciding consensus algorithm with $n \geq 2f_P+1$ and $m \geq 2f_M+1$. However, this is overkill since those algorithms use sophisticated mechanisms (signatures, non-equivocation) to guard against Byzantine behavior. With only crash failures, we now show it is possible to retain the efficiency of a 2-deciding algorithm while improving resiliency. In Section 8.4.1, we first give a 2-deciding algorithm that allows the crash of all but one process ($n \geq f_P+1$) and a minority of memories ($m \geq 2f_M+1$). In Section 8.4.2, we improve resilience further by giving a 2-deciding algorithm that tolerates crashes of a minority of the combined set of memories and processes.

8.4.1 Protected Memory Paxos

Our starting point is the Disk Paxos algorithm [126], which works in a system with processes and memories where $n \geq f_P + 1$ and $m \geq 2f_M + 1$. This is our resiliency goal, but Disk Paxos takes four delays in common executions. Our new algorithm, called Protected Memory Paxos, removes two delays; it retains the structure of Disk Paxos but uses permissions to skip steps. Initially some fixed leader $\ell = p_1$ has exclusive write permission to all memories; if another process becomes leader, it takes the exclusive permission. Having exclusive permission allows a leader ℓ to optimize its execution, because ℓ can do two things simultaneously: (1) write its consensus proposal and (2) determine whether another leader took over. Specifically, if ℓ succeeds in (1), it knows no leader ℓ' took over because ℓ' would have taken the permission. Thus ℓ avoids the last read in Disk Paxos, saving two delays. Of course, care must be taken to implement this without violating safety.

The pseudocode of Protected Memory Paxos is in Algorithm 8.10. Each memory has one memory region, and at any time exactly one process can write to the region. Each memory i holds a register *slot*$[i, p]$ for each process p. Intuitively, *slot*$[i, p]$ is intended for p to write, but p may not have write permission to do that if it is not the leader—in that case, no process writes *slot*$[i, p]$.

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

Algorithm 8.10: Protected Memory Paxos—code for process p

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

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

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

been completed for a majority of the memories. We represent this in the code with a barrier that waits for a majority of the threads.

When a process p becomes leader, it executes the prepare phase (the first leader p_1 can skip this phase in its first execution of the loop), where, for each memory, p attempts to (1) acquire exclusive write permission, (2) write a new proposal number in its slot, and (3) read all slots of that memory. p waits to succeed in executing these steps on a majority of the memories. If any of p's writes fail or p finds a proposal with a higher proposal number, then p gives up. This is represented with an `abort` in the pseudocode; when an `abort` is executed, the for loop terminates. We assume that when the for loop terminates—either because some thread has aborted or because a majority of threads have reached the end of the loop—all threads of the for loop are terminated and control returns to the main loop (lines 11–37).

If p does not abort, it adopts the value with highest proposal number of all those it read in the memories. To make it clear that races should be avoided among parallel threads in the pseudocode, we wrap this part in an `atomic` environment.

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

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

The code ensures that some correct process eventually decides, but it is easy to extend it so all correct processes decide [76], by having a decided process broadcast its decision. Also, the code shows one instance of consensus, with p_1 as initial leader. With many consensus instances, the leader terminates one instance and becomes the default leader in the next.

Correctness of Protected Memory Paxos. We now present the proof of correctness and common-case running time of Algorithm 8.10

We first show that Algorithm 8.10 correctly implements consensus, starting with validity. Intuitively, validity is preserved because each process that writes any value in a slot either writes its own value, or adopts a value that was previously written in a slot. We show that every value written in any slot must have been the input of some process.

Theorem 8.4.1 (Validity)**.** *In Algorithm 8.10, if a process p decides a value v, then v was the input to some process.*

Proof. Assume by contradiction that some process p decides a value v and v is not the input of any process. Since v is not the input value of p, then p must have adopted v by reading it from some process p' at line 23. Note also that a process cannot adopt the initial value, and thus, v must have been written in p''s memory by some other process. Thus we can define a sequence $s_1, s_2, \ldots, s_k$, where s_i adopts v from the location where it was written by s_{i+1} and $s_1 = p$. This sequence is necessarily finite since there have been a finite number of steps taken up to the point

when p decided v. Therefore, there must be a last element of the sequence, s_k who wrote v in line 35 without having adopted v. This implies v was s_k's input value, a contradiction. □

We now focus on agreement.

Theorem 8.4.2 (Agreement)**.** *In Algorithm 8.10, for any processes p and q, if p and q decide values v_p and v_q respectively, then $v_p = v_q$.*

Before showing the proof of the theorem, we first introduce the following useful lemmas.

Lemma 8.4.3. *The values a leader accesses on remote memory cannot change between when it reads them and when it writes them.*

Proof. Recall that each memory only allows write-access to the most recent process that acquired it. In particular, that means that each memory only gives access to one process at a time. Note that the only place at which a process acquires write-permissions on a memory is at the very beginning of its run, before reading the values written on the memory. In particular, for each memory d a process does not issue a read on d before its permission request on d successfully completes. Therefore, if a process p succeeds in writing on memory m, then no other process could have acquired d's write-permission after p did, and therefore, no other process could have changed the values written on m after p's read of m. □

Lemma 8.4.4. *If a leader writes values v_i and v_j at line 35 with the same proposal number to memories i and j, respectively, then $v_i = v_j$.*

Proof. Assume by contradiction that a leader p writes different values $v_1 \neq v_2$ with the same proposal number. Since each thread of p executes the phase 2 write (line 35) at most once per proposal number, it must be that different threads T_1 and T_2 of p wrote v_1 and v_2, respectively. If p does not perform phase 1 (i.e., if $p = p_1$ and this is p's first attempt), then it is impossible for T_1 and T_2 to write different values, since CurrentVal was set to v at line 14 and was not changed afterwards. Otherwise (if p performs phase 1), then let t_1 and t_2 be the times when T_1 and T_2 executed line 8, respectively (T_1 and T_2 must have done so, since we assume that they both reached the phase 2 write at line 35). Assume *wlog* that $t_1 \leq t_2$. Due to the check and abort at line 29, CurrentVal cannot change after t_1 while keeping the same proposal number. Thus, when T_1 and T_2 perform their phase 2 writes (after t_1), CurrentVal has the same value as it did at t_1; it is therefore impossible for T_1 and T_2 to write different values. We have reached a contradiction. □

Lemma 8.4.5. *If a process p performs phase 1 and then writes to some memory m with proposal number b at line 35, then p must have written b to m at line 21 and read from m at line 23.*

Proof. Let T be the thread of p which writes to m at line 35. If phase 1 is performed (i.e., the condition at line 19 is satisfied), then by construction T cannot reach line 35 without first performing lines 21 and 23. Since T only communicates with m, it must be that lines 21 and 23 are performed on m. □

Proof of Theorem 8.4.2. Assume by contradiction that $v_p \neq v_q$. Let b_p and b_q be the proposal numbers with which v_p and v_q are decided, respectively. Let W_p (resp. W_q) be the set of memories to which p (resp. q) successfully wrote in phase 2 line 35 before deciding v_p (resp. v_q). Since

W_p and W_q are both majorities, their intersection must be non-empty. Let m be any memory in $W_p \cap W_q$.

We first consider the case in which one of the processes did not perform phase 1 before deciding (i.e., one of the processes is p_1 and it decided on its first attempt). Let that process be p *wlog*. Further assume *wlog* that q is the first process to enter phase 2 with a value different from v_p. p's phase 2 write on m must have preceded q obtaining permissions from m (otherwise, p's write would have failed due to lack of permissions). Thus, q must have seen p's value during its read on m at line 23, and thus q cannot have adopted its own value. Since q is the first process to enter phase 2 with a value different from v_p, q cannot have adopted any other value than v_p, so q must have adopted v_p. Contradiction.

We now consider the remaining case: both p and q performed phase 1 before deciding. We assume *wlog* that $b_p < b_q$ and that b_q is the smallest proposal number larger than b_p for which some process enters phase 2 with CurrentVal $\neq v_p$.

Since $b_p < b_q$, p's read at m must have preceded q's phase 1 write at m (otherwise p would have seen q's larger proposal number and aborted). This implies that p's phase 2 write at m must have preceded q's phase 1 write at m (by Lemma 8.4.3). Thus q must have seen v_p during its read and cannot have adopted its own input value. However, q cannot have adopted v_p, so q must have adopted v_q from some other slot sl that q saw during its read. It must be that $sl.minProposal < b_q$, otherwise q would have aborted. Since $sl.minProposal \geq sl.accProposal$ for any slot, it follows that $sl.accProposal < b_q$. If $sl.accProposal < b_p$, q cannot have adopted $sl.value$ in line 30 (it would have adopted v_p instead). Thus it must be that $b_p \leq sl.accProposal < b_q$; however, this is impossible because we assumed that b_q is the smallest proposal number larger than b_p for which some process enters phase 2 with CurrentVal $\neq v_p$. We have reached a contradiction. □

Finally, we prove that the termination property holds.

Theorem 8.4.6 (Termination)**.** *Eventually, all correct processes decide.*

Lemma 8.4.7. *If a correct process p is executing the for loop at lines 18–37, then p will eventually exit from the loop.*

Proof. The threads of the for loop perform the following potentially blocking steps: obtaining permission (line 20), writing (lines 21 and 35), reading (line 23), and waiting for other threads (the barrier at line 7 and the exit condition at line 37). By our assumption that a majority of memories are correct, a majority of the threads of the for loop must eventually obtain permission in line 20 and invoke the write at line 21. If one of these writes fails due to lack of permission, the loop aborts and we are done. Otherwise, a majority of threads will perform the read at line 23. If some thread aborts at lines 22 and 26, then the loop aborts and we are done. Otherwise, a majority of threads must add themselves to ListOfReady, pass the barrier at line 7 and invoke the write at line 35. If some such write fails, the loop aborts; otherwise, a majority of threads will reach the check at line 37 and thus the loop will exit. □

Proof of Theorem 8.4.6. The Ω failure detector guarantees that eventually, all processes trust the same correct process p. Let t be the time after which all correct processes trust p forever. By Lemma 8.4.7, at some time $t' \geq t$, all correct processes except p will be blocked at line 12. Therefore, the *minProposal* values of all memories, on all slots except those of p stop increasing.

Thus, eventually, p picks a *propNr* that is larger than all others written on any memory, and stops restarting at line 26. Furthermore, since no process other than p is executing any steps of the algorithm, and in particular, no process other than p ever acquires any memory after time t', p never loses its permission on any of the memories. So, all writes executed by p on any correct memory must return ack. Therefore, p will decide and broadcast its decision to all. All correct processes will receive p's decision and decide as well. □

To complete the proof of Theorem 8.4.9, we now show that Algorithm 8.10 is 2-deciding.

Theorem 8.4.8. *Algorithm 8.10 is 2-deciding.*

Proof. Consider an execution in which p_1 is timely, and no process's failure detector ever suspects p_1. Then, since no process thinks itself the leader, and processes do not deviate from their protocols, no process calls changePermission on any memory. Furthermore, p_1's firstAttempt flag is set, since it never switched leaders. So, since p_1 initially has write permission on all memories, all of p_1's writes succeed. Therefore, p_1 terminates, deciding its own proposed value v, after one write per memory. □

To following theorem summarizes the result.

Theorem 8.4.9. *Consider a message-and-memory model with up to f_P process crashes and f_M memory crashes, where $n \geq f_P + 1$ and $m \geq 2f_M + 1$. There exists a 2-deciding algorithm for consensus.*

8.4.2 Aligned Paxos

We now further enhance the failure resilience. We show that memories and processes are equivalent *agents*, in that it suffices for a majority of the agents (processes and memories together) to remain alive to solve consensus. Our new algorithm, *Aligned Paxos*, achieves this resiliency. To do so, the algorithm relies on the ability to use both the messages and the memories in our model; permissions are not needed. The key idea is to align a message-passing algorithm and a memory-based algorithm to use any majority of agents. We align Paxos [193] and Protected Memory Paxos so that their decisions are coordinated. More specifically, Protected Memory Paxos and Paxos have two phases. To align these algorithms, we factor out their differences and replace their steps with an abstraction that is implemented differently for each algorithm. The result is our *Aligned Paxos* algorithm, which has two phases, each with three steps: *communicate*, *hear back*, and *analyze*. Each step treats processes and memories separately, and translates the results of operations on different agents to a common language. We implement the steps using their analogues in Paxos and Protected Memory Paxos[4].

The pseudocode of Aligned Paxos is shown in Figures 8.11–8.17.

[4]We believe other implementations are possible. For example, replacing the Protected Memory Paxos implementation for memories with the Disk Paxos implementation yields an algorithm that does not use permissions.

Algorithm 8.11: Aligned Paxos

```
1  A=(P, M) is set of acceptors
2  propose(v){
3    resps = [] //prepare empty responses list
4    choose propNr bigger than any seen before
5    for all a in A{
6      cflag = communicate1(a, propNr)
7      resp = hearback1(a)
8      if (cflag){resps.append((a, resp)) } }
9    wait until resps has responses from a majority of A
10   next = analyze1(resps)
11   if (next == RESTART) restart;
12   resps = []
13   for all a in A{
14     cflag = communicate2(a, next)
15     resp = hearback2(a)
16     if (cflag){resps.append((a, resp)) } }
17   wait until resps has responses from a majority of A
18   next = analyze2(resps)
19   if (next == RESTART) restart;
20   decide next; }
```

Algorithm 8.12: Communicate Phase 1—code for process p

```
1  bool communicate1(agent a, value propNr){
2    if (a is memory){
3      changePermission(a, {(R:Π-{p}, W:∅, RW: {p}); //acquire write
         ↪ permission
4      return write(a[p], {propNr, -, -}) }
5    else{
6      send prepare(propNr) to a
7      return true } }
```

Algorithm 8.13: Hear Back Phase 1—code for process p

```
1  value hearback1(agent a){
2    if (a is memory){
3      for all processes q{
4        localInfo[q] = read(a[q]) }
5      return = localInfo}
6    else{
7      return value received from a } }
```

Algorithm 8.14: Analyze Phase 1—code for process p

```
responses is a list of (agent, response) pairs
value analyze1(responses resps){
  for resp in resps {
    if (resp.agent is memory){
      for all slots s in v.info {
        if (s.minProposal > propNr) return RESTART }
      (v, accProposal) = value and accProposal of slot with highest
      accProposal that had a value } }
  return v where (v, accProposal) is the highest accProposal seen in
     ↪ resps.response of all agents }
```

Algorithm 8.15: Communicate Phase 2—code for process p

```
bool communicate2(agent a, value msg){
  if (a is memory){
    return write(a[p], {msg.propNr, msg.propNr, msg.val})}
  else{
    send accepted(msg.propNr, msg.val) to a
    return true } }
```

Algorithm 8.16: Hear Back Phase 2—code for process p

```
value hearback2(agent a){
  if (a is memory){
    return ack }
  else{
    return value received from a } }
```

Algorithm 8.17: Analyze Phase 2—code for process p

```
value analyze2(value v, responses resps){
  if there are at least A/2 + 1 resps such that resp.response==ack{
    return v }
  return RESTART }
```

8.5 Dynamic Permissions are Necessary for Efficient Consensus

In Section 8.4.1, we showed how dynamic permissions can improve the performance of Disk Paxos. Are dynamic permissions necessary? We prove that with shared memory (or disks) alone, one cannot achieve 2-deciding consensus, even if the memory never fails, it has static permissions, processes may only fail by crashing, and the system is partially synchronous in the sense that eventually there is a known upper bound on the time it takes a correct process to take a step [109]. This result applies a fortiori to the Disk Paxos model [126].

Theorem 8.5.1. *Consider a partially synchronous shared-memory model with registers, where registers can have arbitrary static permissions, memory never fails, and at most one processes may fail by crashing. No consensus algorithm is 2-deciding.*

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

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

Theorem 8.5.1, together with the Fast Paxos algorithm of Lamport [196], shows that an atomic read-write shared memory model is strictly weaker than the message passing model in its ability to solve consensus quickly. This result may be of independent interest, since often the classic shared memory and message passing models are seen as equivalent, because of the seminal computational equivalence result of Attiya, Bar-Noy, and Dolev [29]. Interestingly, it is known that shared memory can tolerante more failures when solving consensus (with randomization or partial synchrony) [25, 65], and therefore it seems that perhaps shared memory is strictly stronger than message passing for solving consensus. However, our result shows that there are aspects in which message passing is stronger than shared memory. In particular, message passing can solve consensus faster than shared memory in well-behaved executions.

Chapter 9

Microsecond-Scale State Machine Replication

9.1 Introduction

In the previous two chapters, we studied RDMA primarily through a theoretical lens. We abstracted its features into the the ready-active model and reasoned about the capabilities of this new communication model compared to previous ones. However, when developing a theory meant to reflect practice, it is important to check that the theoretical results can indeed translate back into practical advantages. In this chapter, we present a system built using RDMA to enhance availability in Microsecond-scale computing.

Enabled by modern technologies such as RDMA, Microsecond-scale computing is emerging as a must [37]. A microsecond app might be expected to process a request in 10 microseconds. Areas where software systems care about microsecond performance include finance (e.g., trading systems), embedded computing (e.g., control systems), and microservices (e.g., key-value stores). Some of these areas are critical and it is desirable to replicate their microsecond apps across many hosts to provide high availability, due to economic, safety, or robustness reasons. Typically, a system may have hundreds of microservice apps [127], some of which are stateful and can disrupt a global execution if they fail (e.g., key-value stores)—these apps should be replicated for the sake of the whole system.

The golden standard to replicate an app is State Machine Replication (SMR) [263], whereby replicas execute requests in the same total order determined by a consensus protocol. Unfortunately, traditional SMR systems add hundreds of microseconds of overhead even on a fast network [158]. Recent work explores modern hardware in order to improve the performance of replication [41, 174, 178, 187, 250, 279]. The fastest of these (e.g., Hermes [178], DARE [250], and HovercRaft [187]) induce however an overhead of several microseconds, which is clearly high for apps that themselves take few microseconds. Furthermore, when a failure occurs, prior systems incur a prohibitively large fail-over time in the tens of *milli*seconds (not microseconds). For instance, HovercRaft takes 10 milliseconds, DARE 30 milliseconds, and Hermes at least 150 milliseconds. The rationale for such large latencies are timeouts that account for the natural fluctuations in the latency of modern networks. Improving replication and fail-over latencies

requires fundamentally new techniques.

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

To achieve this performance, Mu introduces a new SMR protocol that fundamentally changes how RDMA can be leveraged for replication. Our protocol, inspired by the one-shot consensus algorithm presented in Chapter 8, reaches consensus and replicates a request with just one round of parallel RDMA write operations on a majority of replicas. This is in contrast to prior approaches, which take multiple rounds [41, 250, 279] or resort to two-sided communication [158, 174, 189, 226]. Roughly, in Mu the leader replicates a request by simply using RDMA to write it to the log of each replica, without additional rounds of communication. Doing this correctly is challenging because concurrent leaders may try to write to the logs simultaneously. In fact, the hardest part of most replication protocols is the mechanism to protect against races of concurrent leaders (e.g., Paxos proposal numbers [193]). Traditional replication implements this mechanism using send-receive communication (two-sided operations) or multiple rounds of communication. Instead, Mu uses RDMA write permissions to guarantee that a replica's log can be written by only one leader. This is similar to the crash-consensus algorithm of Chapter 8. However, critical to the correctness and efficiency of the fully-fledged SMR system are other mechanisms as well, including those to change leaders and garbage collect logs, which we describe in this chapter.

Mu also improves fail-over time to just 873 microseconds, with the 99-th percentile at 945 microseconds, which cuts fail-over time of prior systems by an order of magnitude. The fact that Mu significantly improves both replication overhead and fail-over latency is perhaps surprising: folklore suggests a trade-off between the latencies of replication in the fast path, and fail-over in the slow path.

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

For leader change, the latency comes from the cost of changing RDMA write permissions, which with current NICs are hundreds of microseconds. This is higher than we expected: it is far slower than RDMA reads and writes, which go over the network. We attribute this delay to a lack of hardware optimization. RDMA has many methods to change permissions: (1) re-register

memory regions, (2) change queue-pair access flags, or (3) close and reopen queue pairs. We carefully evaluate the speed of each method and propose a scheme that combines two of them using a fast-slow path to minimize latency. Despite our efforts, the best way to cut this latency further is to improve the NIC hardware.

We implemented Mu and used it to replicate several apps: a financial exchange app called Liquibook [213], Redis, Memcached, and an RDMA-based key-value stored called HERD [170].

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

Mu has some limitations. First, Mu relies on RDMA and so it is suitable only for networks with RDMA, such as local area networks, but not across the wide area. Second, Mu is an in-memory system that does not persist data in stable storage—doing so would add additional latency dependent on the device speed. [1] However, we observe that the industry is working on extensions of RDMA for persistent memory, whereby RDMA writes can be flushed at a remote persistent memory with minimum latency [268]—once available, this extension will provide persistence for Mu.

To summarize, this chapter makes the following contributions:

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

One might argue that Mu is ahead of its time, as most apps today are not yet microsecond apps. However, this situation is changing. We already have important microsecond apps in areas such as trading, and more will come as existing timing requirements become stricter and new systems emerge as the composition of a large number of microservices (Section 9.2.1).

9.2 Background

9.2.1 Microsecond Applications and Computing

Apps that are consumed by humans typically work at the millisecond scale: to the human brain, the lowest reported perceptible latency is 13 milliseconds [251]. Yet, we see the emergence of apps that are consumed not by humans but by other computing systems. An increasing number of such systems must operate at the microsecond scale, for competitive, physical, or composition reasons. Schneider [262] speaks of a microsecond market where traders spend massive resources

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

to gain a microsecond advantage in their high-frequency trading. Industrial robots must orchestrate their motors with microsecond granularity for precise movements [21]. Modern distributed systems are composed of hundreds [127] of stateless and stateful microservices, such as key-value stores, web servers, load balancers, and ad services—each operating as an independent app whose latency requirements are gradually decreasing to the microsecond level [62], as the number of composed services is increasing. With this trend, we already see the emergence of key-value stores with microsecond latency (e.g., [174, 241]).

To operate at the microsecond scale, the computing ecosystem must be improved at many layers. This is happening gradually by various recent efforts. Barroso et al [37] argue for better support of microsecond-scale events. The latest Precision Time Protocol improves clock synchronization to achieve submicrosecond accuracy [23]. And other recent work improves CPU scheduling [62, 246, 253], thread management [254], power management [252], RPC handling [96, 174], and the network stack [246]—all at the microsecond scale. Mu fits in this context, by providing microsecond SMR.

9.2.2 State Machine Replication

State Machine Replication (SMR) replicates a service (e.g., a key-value storage system) across multiple physical servers called *replicas*, such that the system remains available and consistent even if some servers fail. SMR provides strong consistency in the form of linearizability [154]. A common way to implement SMR, which we adopt in this work, is as follows: each replica has a copy of the service software and a log. The log stores client requests. We consider non-durable SMR systems [164, 169, 209, 216, 243, 247], which keep state in memory only, without logging updates to stable storage. Such systems make an item of data reliable by keeping copies of it in the memory of several nodes. Thus, the data remains recoverable as long as there are fewer simultaneous node failures than data copies [250].

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

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

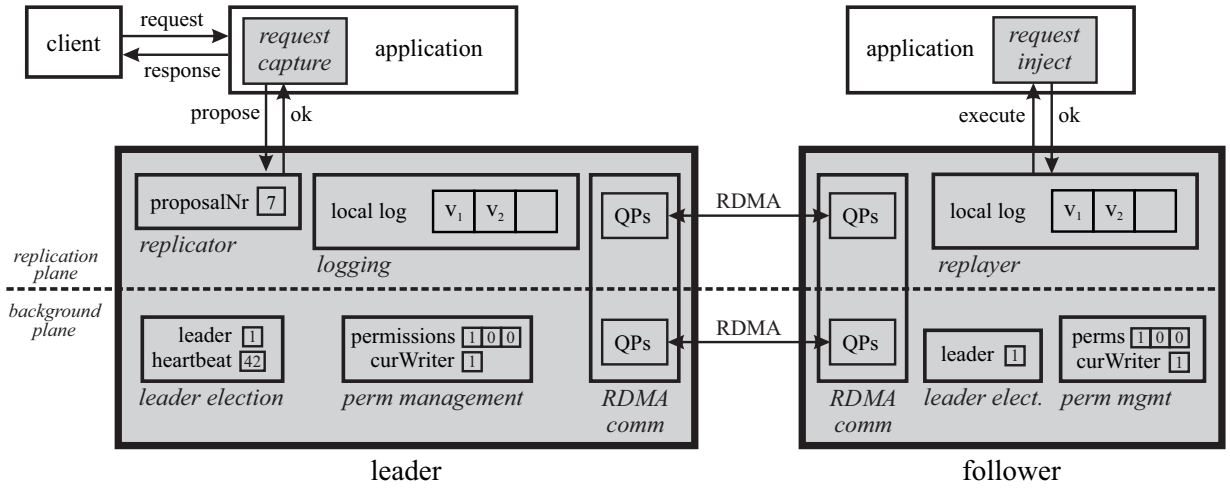


Figure 9.1: Architecture of Mu. Grey color shows Mu components. A replica is either a leader or a follower, with different behaviors. The leader captures client requests and writes them to the local logs of all replicas. Followers replay the log to inject the client requests into the application. A leader election component includes a heartbeat and the identity of the current leader. A permission management component allows a leader to request write permission to the local log while revoking the permission from other nodes.

9.3 Overview of Mu

9.3.1 Architecture

Figure 9.1 depicts the architecture of Mu. At the top, a client sends requests to an application and receives a response. We are not particularly concerned about how the client communicates with the application: it can use a network, a local pipe, a function call, etc. We do assume however that this communication is amenable to being captured and injected. That is, there is a mechanism to capture requests from the client before it reaches the application, so we can forward these requests to the replicas; a request is an opaque buffer that is not interpreted by Mu. Similarly, there is a mechanism to inject requests into the app. Providing such mechanisms requires changing the application; however, in our experience, the changes are small and non-intrusive. These mechanisms are standard in any SMR system.

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

The main challenge in the design of SMR protocols is to handle leader failures. Of particular

concern is the case when a leader appears failed (due to intermittent network delays) so another leader takes over, but the original leader is still active.

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

The new leader revokes a write permission by any old leaders, thereby ensuring that old leaders cannot interfere with the operation of the new leader [13]. The new leader also reconstructs any partial work left by prior leaders.

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

The replication plane has three components:

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

The background plane has two components:

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

We describe these planes in more detain in Section 9.4 and Section 9.5.

9.3.2 RDMA Communication

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

Each replica also maintains two MRs, one for each plane. The MR of the replication plane contains the consensus log and the MR of the background plane contains metadata for leader election (Section 9.5.1) and permission management (Section 9.5.2). During execution, replicas may change the level of access to their log that they give to each remote replica; this is done

by changing QP access flags. Note that all replicas always have remote read and write access permissions to the memory region in the background plane of each replica.

9.4 Replication Plane

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

In this section, we discuss algorithmic details related to replication in Mu. For pedagogical reasons, we first describe a basic version of the algorithm and then discuss extensions and optimizations to improve functionality and performance. In this section, we give intuition for the correctness of our algorithm.

9.4.1 Basic Algorithm

The leader captures client requests, and calls *propose* to replicate these requests. It is simplest to understand our replication algorithm relative to the Paxos algorithm, which we briefly summarize; for details, we refer the reader to [193]. In Paxos, for each slot of the log, a leader first executes a *prepare phase* where it sends a proposal number to all replicas.[2] A replica replies with either nack if it has seen a higher proposal number, or otherwise with the value with the highest proposal number that it has accepted. After getting a majority of replies, the leader adopts the value with the highest proposal number. If it got no values (only acks), it adopts its own proposal value. In the next phase, the *accept phase*, the leader sends its proposal number and adopted value to all replicas. A replica acks if it has not received any prepare phase message with a higher proposal number.

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

However, this by itself is not enough; Paxos relies on active participation from the followers not only for the data itself, but also to avoid races. Simply publishing the relevant data on each replica is not enough, since two competing leaders could miss each other's updates. This can be avoided if each of the leaders rereads the value after writing it [126]. However, this requires more communication. To avoid this, we shift the focus from the communication itself to the *prevention* of bad communication. A leader ℓ maintains a set of *confirmed followers*, which have granted write permission to ℓ and revoked write permission from other leaders before ℓ begins its operation. This is what prevents races among leaders in Mu. We describe these mechanisms in more detail below.

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

Algorithm 9.2: Log Structure

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

Log Structure. The main data structure used by the algorithm is the consensus log kept at each replica (Algorithm 9.2). The log consists of (1) a *minProposal* number, representing the smallest proposal number with which a leader may enter the accept phase on this replica; (2) a *first undecided offset (FUO)*, representing the lowest log index which this replica believes to be undecided; and (3) a sequence of slots—each slot is a $(propNr, value)$ tuple.

Algorithm Description. Each leader begins its propose call by constructing its *confirmed followers* set (Algorithm 9.3, lines 8–11). This step is only necessary the first time a new leader invokes propose or immediately after an abort. This step is done by sending permission requests to all replicas and waiting for a majority of acks. When a replica acks, it means that this replica has granted write permission to this leader and revoked it from other replicas. The leader then adds this replica to its confirmed followers set. During execution, if the leader ℓ fails to write to one of its confirmed followers, because that follower crashed or gave write access to another leader, ℓ aborts and, if it still thinks it is the leader, it calls propose again.

After establishing its confirmed followers set, the leader invokes the prepare phase. To do so, the leader reads the *minProposal* from its confirmed followers (line 18) and chooses a proposal number propNum which is larger than any that it has read or used before. Then, the leader writes its proposal number into *minProposal* for each of its confirmed followers. Recall that if this write fails at any follower, the leader aborts. It is safe to overwrite a follower f's *minProposal* in line 21 because, if that write succeeds, then ℓ has not lost its write permission since adding f to its confirmed followers set, meaning no other leader wrote to f since then. To complete its prepare phase, the leader reads the relevant log slot of all of its confirmed followers and, as in Paxos, adopts either (a) the value with the highest proposal number, if it read any non-$\perp$ values, or (b) its own initial value, otherwise.

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

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

Algorithm 9.3: Basic Replication Algorithm of Mu

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

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

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

9.4.2 Extensions

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

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

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

Omitting the Prepare Phase. Once a leader finds only empty slots at a given index at all of its confirmed followers at line 22, then no higher index may contain an accepted value at any confirmed follower; thus, the leader may omit the prepare phase for higher indexes (until it aborts, after which the prepare phase becomes necessary again). This optimization concerns performance on the common path. With this optimization, the cost of a Propose call becomes a single RDMA write to a majority in the common case.

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

Replayer. Followers continually monitor the log for new entries. This creates a challenge: how to ensure that the follower does not read an incomplete entry that has not yet been fully written

by the leader. We adopt a standard approach: we add an extra *canary byte* at the end of each log entry [214, 279]. Before issuing an RDMA Write to replicate a log entry, the leader sets the entry's canary byte to a non-zero value. The follower first checks the canary and then the entry contents. In theory, it is possible that the canary gets written before the other contents under RDMA semantics. In practice, however, NICs provide left-to-right semantics in certain cases (e.g., the memory region is in the same NUMA domain as the NIC), which ensures that the canary is written last. This assumption is made by other RDMA systems [106, 107, 170, 214, 279]. Alternatively, we could store a checksum of the data in the canary, and the follower could read the canary and wait for the checksum to match the data.

9.5 Background Plane

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

9.5.1 Leader Election

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

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

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

9.5.2 Permission Management

The permission management module is used when changing leaders. Each replica maintains the invariant that only one replica at a time has write permission on its log. As explained in Section 2.4, when a leader changes in Mu, the new leader must request write permission from all the other replicas; this is done through a simple RDMA Write to a *permission request array* on the remote side. When a replica r sees a *permission request* from a would-be leader ℓ, r revokes write access from the current holder, grants write access to ℓ, and sends an ack to ℓ.

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

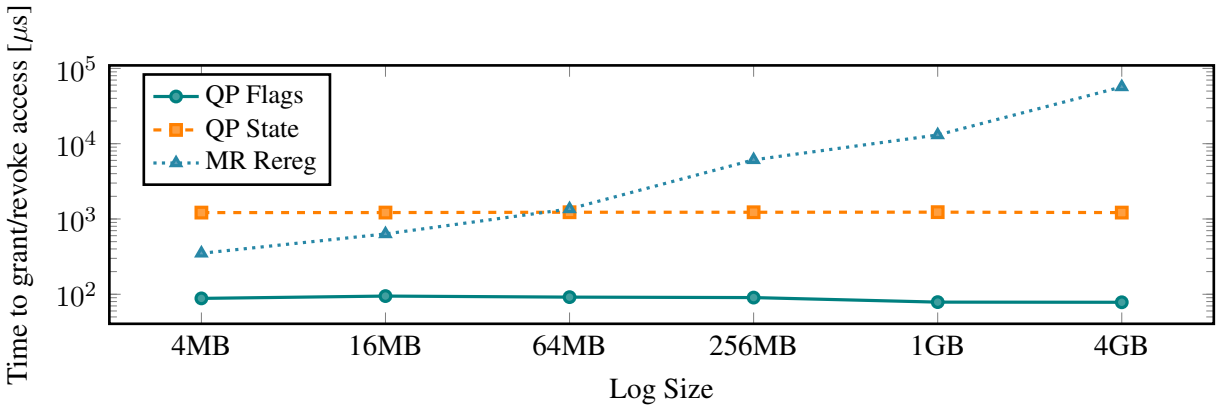


Figure 9.4: Performance comparison of different permission switching mechanisms. *QP Flags*: change the access flags on a QP; *QP Restart*: cycle a QP through the *reset*, *init*, *RTR* and *RTS* states; *MR Rereg*: re-register an RDMA MR with different access flags.

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

We compare the performance of these three mechanisms in Figure 9.4, as a function of the log size (which is the same as the RDMA MR size). We observe that the time to re-register an RDMA MR grows with the size of the MR, and can reach values close to 100ms for a log size of 4GB. On the other hand; the time to change a QPs access flags or cycle it through different states is independent of the MR size, with the former being roughly 10 times faster than the latter. However, changing a QPs access flags while RDMA operations to that QP are in flight sometimes causes the QP to go into an error state. Therefore, in Mu we use a fast-slow path approach: we first optimistically try to change permissions using the faster QP access flag method and, if that leads to an error, switch to the slower, but robust, QP state method.

9.5.3 Log Recycling

Conceptually, a log is an infinite data structure but in practice we need to implement a circular log with limited memory. This is done as follows. Each follower has a local *log head* variable, pointing to the first entry not yet executed in its copy of the application. The replayer thread advances the log head each time it executes an entry in the application. Periodically, the leader's background plane reads the log heads of all followers and computes *minHead*, the minimum of all log head pointers read from the followers. Log entries up to the minHead can be reused. Before these entries can be reused, they must be zeroed out to ensure the correct function of the canary byte mechanism. Thus, the leader zeroes all follower logs after the leader's first undecided offset and before minHead, using an RDMA Write per follower. Note that this means that a new

leader must first execute all leader change actions, ensuring that its first undecided offset is higher than all followers' first undecided offsets, before it can recycle entries. To facilitate the implementation, we ensure that the log is never completely full.

9.5.4 Adding and Removing Replicas

Mu adopts a standard method to add or remote replicas: use consensus itself to inform replicas about the change [193]. More precisely, there is a special log entry that indicates that replicas have been removed or added. Removing replicas is easy: once a replica sees it has been removed, it stops executing, while other replicas subsequently ignore any communication with it. Adding replicas is more complicated because it requires copying the state of an existing replica into the new one. To do that, Mu uses the standard approach of check-pointing state, and we do that from one of the followers [279].

9.6 Implementation

Mu is implemented in 7157 lines of C++ code (CLOC [97]). It uses the *ibverbs* library for RDMA over Infiniband. We implemented all features and extensions in sections 9.4 and 9.5, except adding/removing replicas. Moreover, we implement some standard RDMA optimizations to reduce latency. RDMA Writes and Sends with payloads below a device-specific limit (256 bytes in our setup) are inlined, meaning that their payload is written directly to their work request. We pin threads to cores in the NUMA node of the NIC.

9.7 Evaluation

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

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

We evaluate Mu on a 4-node cluster, where each node has two Intel Xeon E5-2640 v4 CPUs @ 2.40GHz (20 cores, 40 threads total per node), 256 GB of RAM equally split across the two NUMA domains, and a Mellanox Connect-X 4 NIC. All 4 nodes are connected to a single 100 Gbps Mellanox MSB7700 switch through 100 Gbps Infiniband. All experiments show 3-way replication, which accounts for most real deployments [158]. With more replicas, replication latencies increases gradually with the number of replicas, up to 35% higher for Mu (for 9 replicas) and a larger increase than Mu for other systems at every replication level.

We compare against APUS [279], DARE [250], and Hermes [178] where possible. The most recent system, HovercRaft [187], also provides SMR but its latency at 30–60 microseconds is

substantially higher than the other systems, so we do not consider it further. For a fair comparison, we disable APUS's persistence to stable storage, since Mu, DARE, and Hermes all provide only in-memory replication.

We measure time using the POSIX `clock_gettime` function, with the `CLOCK_MONOTONIC` parameter. In our deployment, the resolution and overhead of `clock_gettime` is around $16ns$ [105]. In our figures, we show bars labeled with the median latency, with error bars showing 99-percentile and 1-percentile latencies. These statistics are computed over 1 million samples with a payload of 64-bytes each, unless otherwise stated.

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

The key-value stores that we replicate with Mu are Redis [256], Memcached [229], and HERD [170]. For the first two, the client is assumed to be on a different cluster, and connects to the servers over TCP. In contrast, HERD is a microsecond-scale RDMA-based key-value store. We replicate it over RDMA and use it as an example of a microsecond application. Integration with the three applications requires 183, 228 and 196 additional lines of code, respectively.

The other app is in the context of financial exchanges, in which parties unknown to each other submit buy and sell orders of stocks, commodities, derivatives, etc. At the heart of a financial exchange is an order matching engine [22], such as Liquibook [213], which is responsible for matching the buy and sell orders of the parties. We use Mu to replicate Liquibook. Liquibook's input are buy and sell orders. We created an unreplicated client-server version of Liquibook using eRPC [174], and then replicated this system using Mu. The eRPC integration and the replication required 611 lines of code in total.

9.7.1 Common-Case Latency

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

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

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

Figure 9.5 compares standalone to attached runs as we vary payload size. Liquibook and Herd allow only one payload size (32 and 50 bytes), so they have only one bar each in the graph, while Redis and Memcached have many bars.

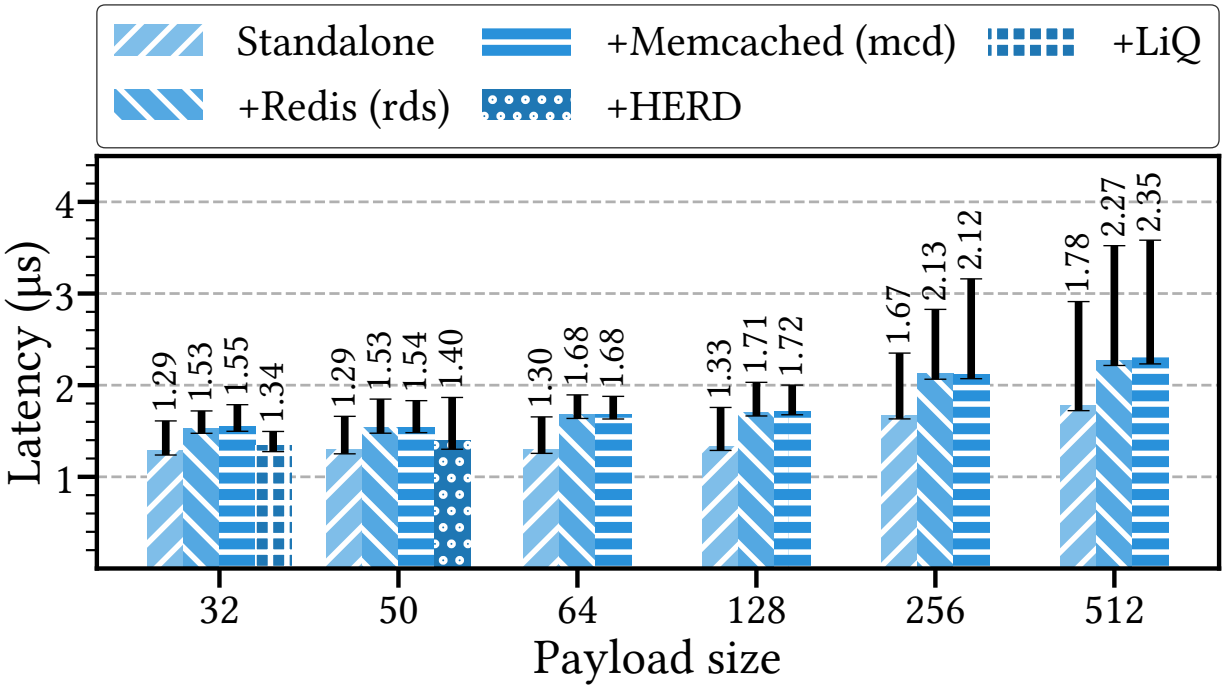


Figure 9.5: Replication latency of Mu integrated into different applications [Memcached (mcd), Liquibook (LiQ), Redis (rds), HERD] and payload sizes.

We see that the standalone version slightly outperforms the attached runs, for all tested applications and payload sizes. This is due to processor cache effects; in standalone runs, replication state, such as log and queue pairs, are always in cache, and the requests themselves need not be fetched from memory. This is not the case when attaching to an application. Mu supports two ways of attaching to an application, which have different processor cache sharing. The *direct* mode uses the same thread to run both the application and the replication, and so they share L1 and L2 caches. In contrast, the *handover* method places the application thread on a separate core from the replication thread, thus avoiding sharing L1 or L2 caches. Because the application must communicate the request to the replication thread, the handover method requires a cache coherence miss per replicated request. This method consistently adds ≈400ns over the standalone method. For applications with large requests, this overhead might be preferable to the one caused by the direct method, where replication and application compete for CPU time. For lighter weight applications, the direct method is preferable. In our experiments, we measure both methods and show the best method for each application: Liquibook and HERD use the direct method, while Redis and Memcached use the handover method.

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

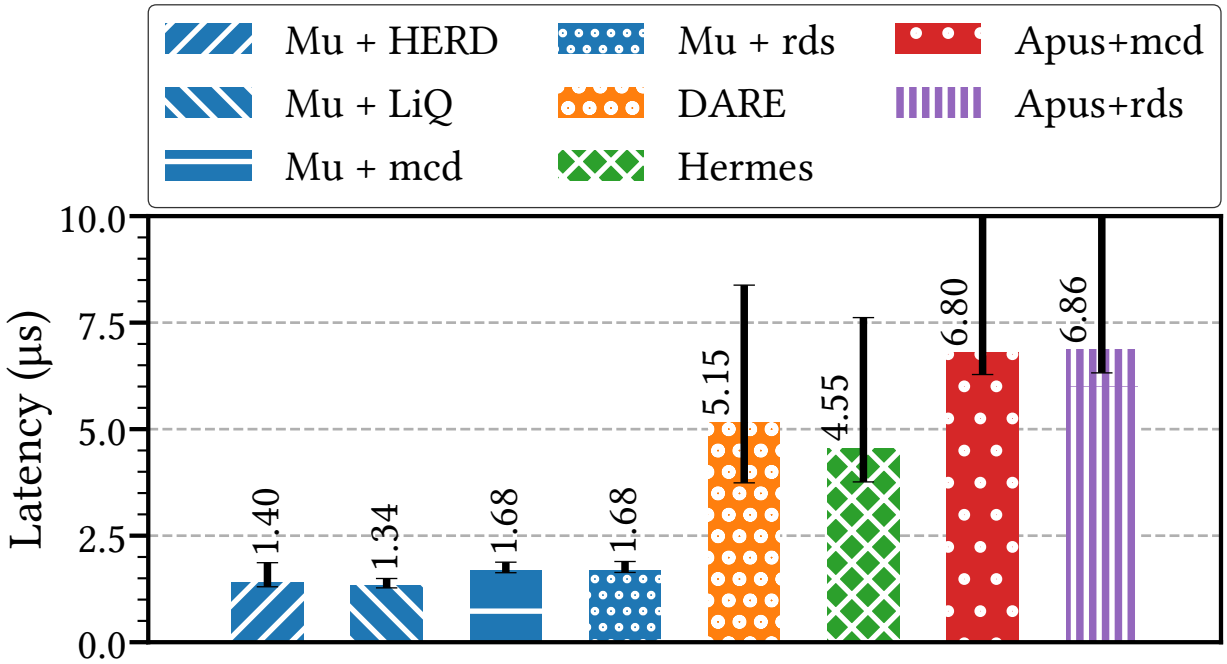


Figure 9.6: Replication latency of Mu compared with other replication solutions: DARE, Hermes, Apus on memcached (mcd), and Apus on Redis (rds).

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

Figure 9.6 shows the replication latencies of these systems. Mu's median latency outperforms all competitors by at least $2.7\times$, outperforming APUS on the same applications by $4\times$. Furthermore, Mu has smaller tail variation, with a difference of at most 500ns between the 1-percentile and 99-percentile latency. In contrast, Hermes and DARE both varied by more than $4\mu s$ across our experiments, with APUS exhibiting 99-percentile executions up to $20\mu s$ slower (cut off in the figure). We attribute this higher variance to two factors: the need to involve the CPU of many replicas in the critical path (Hermes and APUS), and sequentializing several RDMA operations so that their variance aggregates (DARE and APUS).

End-to-End Latencies. Figure 9.7 shows the end-to-end latency of our tested applications, which includes the latency incurred by the application and by replication (if enabled). We show the result in three graphs corresponding to three classes of applications.

The leftmost graph is for Liquibook. The left bar is the unreplicated version, and the right bar is replicated with Mu. We can see that the median latency of Liquibook without replication is $4.08\mu s$, and therefore the overhead of replication is around 35%. There is a large variance in

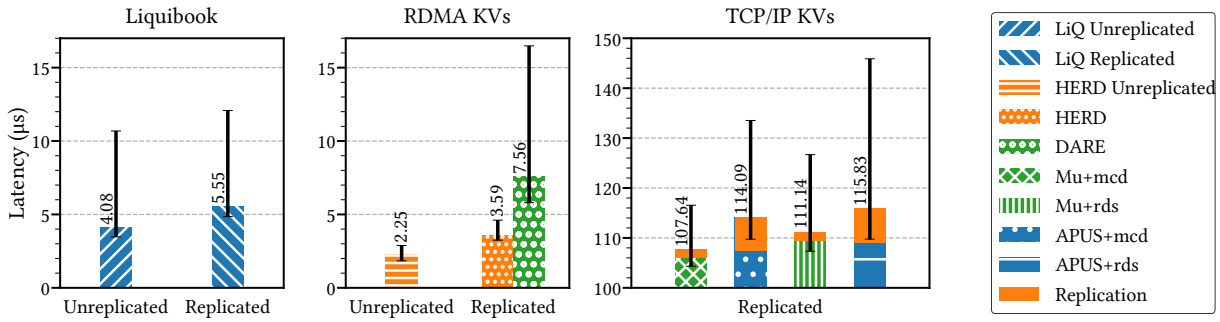


Figure 9.7: End-to-end latencies of applications. The first graph show a financial exchange app (Liquibook) unreplicated and replicated with Mu. The second graph shows microsecond key-value stores: HERD unreplicated, HERD replicated with Mu, and DARE. The third graph shows traditional key-value stores: Memcached and Redis, replicated with Mu and APUS. In this graph, each bar is split in two parts: application latency (bottom) and replication latency (top).

latency, even in the unreplicated system. This variance comes from the client-server communication of Liquibook, which is based on eRPC. This variance changes little with replication. The other replication systems cannot replicate Liquibook (as noted before, DARE and Hermes are bolted onto their app, and APUS can replicate only socket-based applications). However, extrapolating their latency from Figure 9.6, they would add unacceptable overheads—over 100% overhead for the best alternative (Hermes).

The middle graph in Figure 9.7 shows the client-to-client latency of replicated and unreplicated microsecond-scale key-value stores. The first bars in orange shows HERD unreplicated and HERD replicated by Mu. The green bar shows DARE's key-value store with its own replication system. The median unreplicated latency of HERD is $2.25\mu s$, and Mu adds $1.34\mu s$. While this is a significant overhead (59% of the original latency), this overhead is lower than any alternative. We do not show Hermes in this graph since Hermes does not allow for a separate client, and only generates its requests on the servers themselves. HERD replicated with Mu is the best option for a replicated key-value store, with overall median latency $2\times$ lower than the next best option, with a much lower variance.

The rightmost graph in Figure 9.7 shows the replication of the traditional key-value stores, Redis and Memcached. For these applications, we compare replication with Mu to replication with APUS. Each bar has two parts: the bottom is the latency of the application and client-server communication, and the top is the replication latency. Note that the scale starts at $100\mu s$ to show better precision.

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

9.7.2 Fail-Over Time

We now study Mu's fail-over time. In these experiments, we run the system and subsequently introduce a leader failure. To get a thorough understanding of the fail-over time, we repeatedly introduce leader failures (1000 times) and plot a histogram of the fail-over times we observe. We

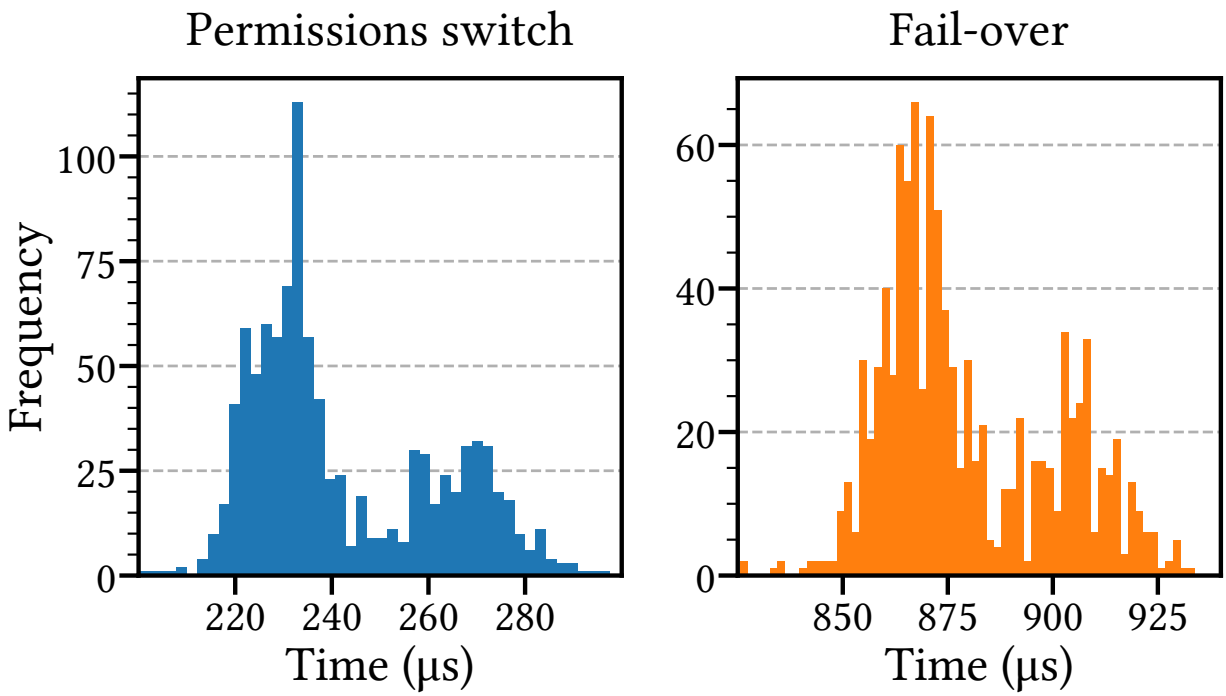


Figure 9.8: Fail-over time distribution.

also time the latency of permission switching, which corresponds to the time to change leaders after a failure is detected. The detection time is the difference between the total fail-over time and the permission switch time.

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

Figure 9.8 shows the results. We first note that the total fail-over time is quite low; the median fail-over time is $873\mu s$ and the 99-percentile fail-over time is $947\mu s$, still below a millisecond. This represents an order of magnitude improvement over the best competitor at $\approx$10 ms (HovercRaft [187]).

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

The rest of the fail-over time is attributed to failure detection ($\approx 600\mu s$). Although our pull-score mechanism does not rely on network variance, there is still variance introduced by process scheduling (e.g., in rare cases, the leader process is descheduled by the OS for tens of microseconds)—this is what prevented us from using smaller timeouts/scores and it is an area under active investigation for microsecond apps [62, 246, 253, 254].

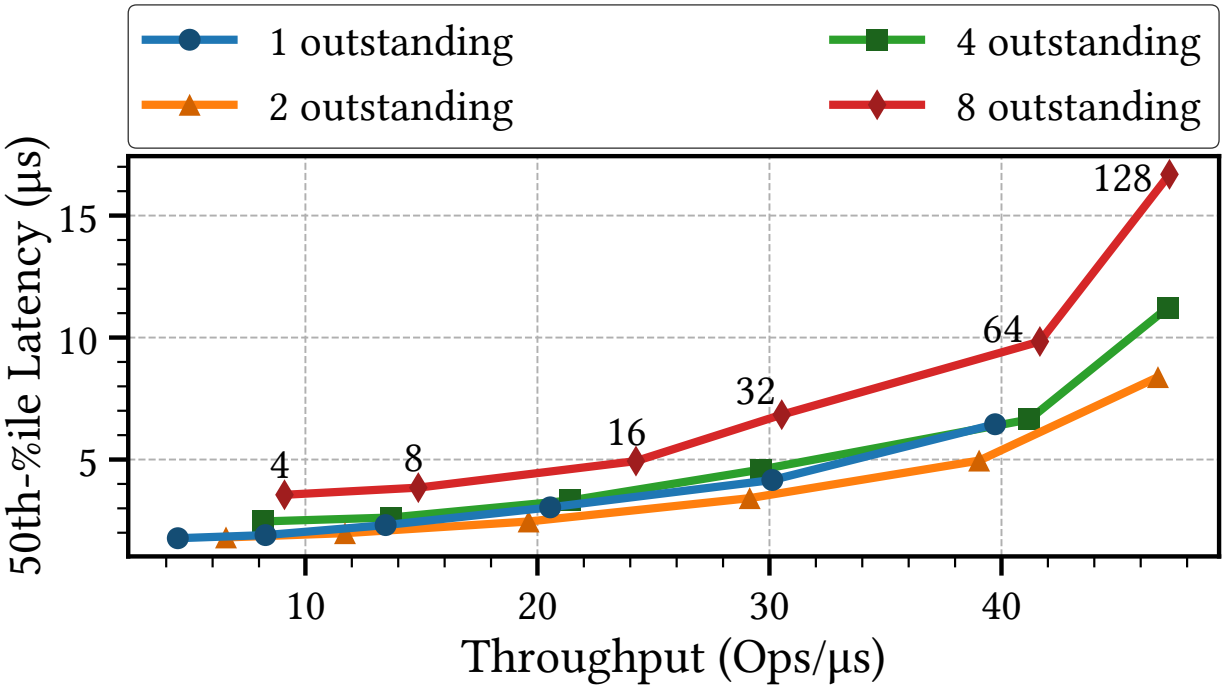


Figure 9.9: Latency vs throughput. Each line represents a different number of allowed concurrent outstanding requests. Each point on the lines represents a different batch size. Batch size shown as annotation close to each point.

9.7.3 Throughput

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

Figure 9.9 shows the results in a latency-throughput graph. Each line represents a different max number of outstanding requests, and each data point represents a different batch size. As before, we use 64-byte requests.

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

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

There is a throughput wall at around 45 Ops/μs, with latency rising sharply. This can be traced to the transition between the client requests and the replication protocol at the leader replica. The leader must copy the request it receives into a memory region prepared for its RDMA write. This memory operation becomes a bottleneck. We could optimize throughput further by allowing direct contact between the client and the follower replicas. However, that

may not be useful as the application itself might need some of the network bandwidth for its own operation, so the replication protocol should not saturate the network.

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

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

9.8 Related Work

SMR in General. State machine replication is a common technique for building fault-tolerant, highly available services [193, 263]. Many practical SMR protocols have been designed, addressing simplicity [61, 67, 158, 198, 242], cost [189, 196], and harsher failure assumptions [72, 73, 126, 189]. In the original scheme, which we follow, the order of all operations is agreed upon using consensus instances. At a high-level, our Mu protocol resembles the classical Paxos algorithm [193], but there are some important differences. In particular, we leverage RDMA's ability to grant and revoke access permissions to ensure that two leader replicas cannot both write a value without recognizing each other's presence. This allows us to optimize out participation from the follower replicas, leading to better performance. Furthermore, these dynamic permissions guide our unique leader changing mechanism.

Several implementations of multipaxos avoid repeating Paxos's prepare phase for every consensus instance, as long as the same leader remains [78, 194, 226]. Piggybacking a commit message onto the next replicated request, as is done in Mu, is also used as a latency-hiding mechanism in [226, 279].

Aguilera et al. [12] suggested the use of local heartbeats in a leader election algorithm designed for a theoretical message-and-memory model, in an approach similar to our pull-score mechanism. However, no system has so far implemented such local heartbeats for leader election in RDMA.

Single round-trip replication has been achieved in several previous works using two-sided sends and receives [108, 178, 180, 189, 196]. Theoretical work has shown that single-shot consensus can be achieved in a single one-sided round trip [13]. However, Mu is the first system to put that idea to work and implement one-sided single round trip SMR.

Alternative reliable replication schemes totally order only non-conflicting operations [86, 156, 178, 195, 247, 249, 265]. These schemes require opening the service being replicated to identify which operations commute. In contrast, we designed Mu assuming the replicated service is a black box. If desired, several parallel instances of Mu could be used to replicate concurrent operations that commute. This could be used to increase throughput in specific applications.

It is also important to notice that we consider "crash" failures. In particular, we assume nodes cannot behave in a Byzantine manner [72, 84, 189].

Improving the Stack Underlying SMR. While we propose a new SMR algorithm adapted to RDMA in order to optimize latency, other systems keep a classical algorithm but improve the underlying communication stack [174, 208]. With this approach, somehow orthogonal to ours, the best reported replication latency is 5.5 μs [174], almost $5\times$ slower than Mu. HovercRaft [187] shifts the SMR from the application layer to the transport layer to avoid IO and CPU bottlenecks on the leader replica. However, their request latency is more than an order of magnitude more than that of Mu, and they do not optimize fail-over time.

Some SMR systems leverage recent technologies such as programmable switches and NICs [164, 169, 209, 216]. However, programmable networks are not as widely available as RDMA, which has been commoditized with technologies such as RoCE and iWARP.

SMR over RDMA. A few SMR systems have recently been designed for RDMA [41, 250, 279]. DARE [250] was the first RDMA-based SMR system. Similarly to Mu, DARE uses only one-sided RDMA verbs executed by the leader to replicate the log in normal execution. However, DARE requires updating the tail pointer of each replica's log in a separate RDMA Write from the one that copies over the new value, and therefore induces more round-trips for replication. Furthermore, DARE has a heavier leader election protocol than Mu's. DARE's leader election is similar to the one used in RAFT, in which care is taken to ensure that at most one process considers itself leader at any point in time. APUS [279] improved upon DARE's throughput. However, APUS requires active participation from the follower replicas during the replication protocol, resulting in higher latencies. Both DARE and APUS use transitions through queue pair states to allow or deny RDMA access. This is reminiscent of our permissions approach, but is less fine grained.

Derecho [41] provides durable and non-durable SMR, by combining a data movement protocol (SMC or RDMC) with a shared-state table primitive (SST) for determining when it is safe to deliver messages. This design yields high throughput but also high latency: a minimum of $10\mu s$ for non-durable SMR [41, Figure 12(b)] and more for durable SMR. This latency results from a node delaying the delivery of a message until all nodes have confirmed its receipt using the SST, which takes additional RDMA communication steps compared to Mu. It would be interesting to explore how Mu's protocol could improve a large system like Derecho.

Other RDMA Applications. More generally, RDMA has recently been the focus of many data center system designs, including key-value stores [106, 170] and transactions [173, 280]. Kalia et al. provide guidelines on the best ways to use RDMA to enhance performance [172]. Many of their suggested optimizations are employed by Mu. Kalia et al. also advocate the use of two-sided RDMA verbs (Sends/Receives) instead of RDMA Reads in situations in which a single RDMA Read might not suffice. However, this does not apply to Mu, since we know a priori which memory location should be read, and we rarely have to follow up with another read.

Failure Detection. Failure detection is typically done using timeouts. Conventional wisdom is that timeouts must be large, in the seconds [205], though some systems report timeouts as low as 10 milliseconds [187]. It is possible to improve detection time using inside information [205, 206] or fine-grained reporting [207], which requires changes to apps and/or the infrastructure. This is orthogonal to our score-based mechanism and could be used to further improve Mu.

9.9 Conclusion

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

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

Chapter 10

Conclusion

In this thesis, we bridge the gap between theory and practice in concurrent and distributed computing, by building and strengthening the theoretical foundations of the study of practical systems. We take two broad approaches toward this goal; firstly, we refine the classic shared-memory model, and secondly, we study and model new technologies. However, plenty of work remains in the effort to close the gap between theory and practice in this field, with many promising directions and open questions arising from the work in this thesis.

Analyzing Shared-Memory Algorithms. Our tool for studying memory schedulers, Severus, along with the experiments presented in Chapter 3, revealed complicated patterns that changed depending on many factors. However, some patterns persisted across the different workloads and architectures tested; we believe that such patterns can be abstracted into a more accurate model of memory-operation schedulers. In particular, a promising direction to take is to make use of Severus to extract a stochastic model that will approximate the observed scheduler behavior. Since the schedules produced by different workloads on different machines can vary greatly and depend on many parameters, we believe that machine learning can aid in creating faithful stochastic models. Plugging such a stochastic model into the modular framework presented in Chapter 2 could yield a model that reflects practice better than before, and can still account for real-world phenomena like the practical success of back-off.

Non-Volatile Memories. The study of non-volatile main memories is relatively young. Unsurprisingly, as is the expectation when a new problem is tackled, several different correctness criteria have been suggested to capture the meaning of persistence [8, 30, 56, 59, 122, 166]. Currently, very little is understood about the comparative power of these different definitions. Do some of the correctness conditions in the literature imply others? Are they incomparable? Does one correctness condition admit solutions that are provably more efficient than another? And, perhaps most importantly, do some of these conditions better capture the desired behavior of practical persistent code? We believe that answering these questions can help streamline the study of NVRAMs, and help us gain clarity in what should or should not be expected from persistent algorithms upon recovery.

On a more practical note, we believe that NVRAM has the most potential to make an impact on applications that already rely on persistence (through disk rather than main memory). Such

applications include large databases and file systems, which must store large amounts of data and be accessible for long periods of time. When we discussed the performance of our persistent simulations in Chapter 5, we viewed our goal as minimizing the *overheads* that we introduce to gain persistence. This may give the impression that we must trade off performance for persistence. However, in applications in which persistence is essential, and which therefore currently rely on disk for persistence, using NVRAM-based solutions could potentially improve performance by orders of magnitude. Applying NVRAM to file systems and databases is already an active area of research [74, 92, 202, 219, 278]. However, many interesting open problems remain in identifying the best data structure for the job, and integrating these applications with persistent data structures that, due to being on main memory rather than disk, can support significantly more concurrent accesses than they could before.

RDMA and Consensus. In Chapter 7, we demonstrate that RDMA can be used sparingly to yield consensus algorithms that are highly scalable and tolerate more failures than is possible in classic networks. In Chapter 8, we prove that RDMA can also be used to achieve impressive common-case performance. However, the latter result does not apply to the large networks considered in the former case. An interesting open problem is to understand whether this disparity is inherent, or whether we can design a highly fault-tolerant RDMA-based algorithm that both scales to large networks, and maintains the same common-case performance. If this is impossible, what is the tradeoff between scalability, fault tolerance, and performance in RDMA networks?

Furthermore, we only briefly touched upon the potential of employing RDMA to tolerate Byzantine failures. In Chapter 8, we proved that it is possible to tolerate $f < n/2$ Byzantine failures when solving consensus in the permissioned M&M model. However, our solution is far from being a practical one; while it boasts optimal common-case performance, its running time quickly deteriorates once network conditions are less than ideal. Furthermore, when building a system that promises to withstand Byzantine attacks, one must ensure that no side-channel attacks are possible. This means not only relying on the theoretical model, but also studying the security of the permission mechanism of RDMA.

Finally, the insights gained in this thesis about RDMA have only focused on the problem of consensus. However, we believe that RDMA can also be used to strengthen solutions to other problems. In fact, some work has already studied the potential of RDMA (and, in particular, the M&M model) to solve leader election [12] and implementing global shared registers [139]. It would be interesting to discover the power of RDMA in other problems as well.

Other Directions. We believe that it is important to continue to narrow the gap between theory and practice in concurrent and distributed systems. Pursuing this long-term goal can take many forms, including studying the relationships between different hardware, and applying tools from different subfields of computer science to help find clean patterns in complicated real systems and apply new theoretical insights in practice.

Bibliography

[1] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed computing (DIST)*, 18(5): 387–408, 2006.

[2] Karl Abrahamson. On achieving consensus using a shared memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 291–302, August 1988.

[3] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[4] Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. Dag-calculus: A calculus for parallel computation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2016.

[5] Umut A Acar, Naama Ben-David, and Mike Rainey. Contention in structured concurrency: Provably efficient dynamic non-zero indicators for nested parallelism. *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.

[6] Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 47–58, August 1992.

[7] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *International Symposium on Computer Architecture (ISCA)*, 1989.

[8] Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241*, 2003.

[9] Marcos K. Aguilera and Sam Toueg. The correctness proof of Ben-Or's randomized consensus algorithm. *Distributed Computing*, 25(5):371–381, October 2012.

[10] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Symposium on Cloud Computing (SoCC)*, pages 121–127, September 2017.

[11] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *USENIX Annual Technical Conference (ATC)*, July 2018.

[12] Marcos K Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–60. ACM, 2018.

[13] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of RDMA on agreement. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 409–418, 2019.

[14] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, Athansios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[15] Dan Alistarh, Thomas Sauerwald, and Milan Vojnović. Lock-free algorithms under stochastic schedulers. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 251–260. ACM, 2015.

[16] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? *Journal of the ACM (JACM)*, 63(4):31, 2016.

[17] Noga Alon, Michael Merritt, Omer Reingold, Gadi Taubenfeld, and Rebecca N Wright. Tight bounds for shared memory systems accessed by byzantine processes. *Distributed computing (DIST)*, 18(2):99–109, 2005.

[18] Cristiana Amza, Alan L. Cox, Shandya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[19] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1990.

[20] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[21] Anonymous. When microseconds count: Fast current loop innovation helps motors work smarter, not harder. `http://e2e.ti.com/blogs_/b/thinkinnovate/archive/2017/11/14/when-microseconds-count-fast-current-loop-innovation-helps-motors-work-smarter-not-harder`.

[22] anonymous. Order matching system. `https://en.wikipedia.org/wiki/Order_matching_system`.

[23] Anonymous. 1588-2019—ieee approved draft standard for a precision clock synchronization protocol for networked measurement and control systems. `https://standards.ieee.org/content/ieee-standards/en/standard/1588-2019.html`.

[24] Krste Asanovic and David Patterson. FireBox: A hardware building block for 2020 warehouse-scale computers. In *USENIX Conference on File and Storage Technologies (FAST)*, February 2014.

[25] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of algorithms*, 11(3):441–461, September 1990.

[26] Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. Analyzing the performance of lock-free data structures: A conflict-based model. In *International Symposium on Distributed*

Computing (DISC), pages 341–355. Springer, 2015.

[27] Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. How lock-free data structures perform in dynamic environments: Models and analyses. *arXiv preprint arXiv:1611.05793*, 2016.

[28] Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *Journal of the ACM (JACM)*, 55(5):20, October 2008.

[29] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, January 1995.

[30] Hagit Attiya, Ohad Ben Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2018.

[31] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4):12, 2015.

[32] Yonatan Aumann and Michael A Bender. Efficient asynchronous consensus with the value-oblivious adversary scheduler. pages 622–633. Springer, 1996.

[33] Baruch Awerbuch, Andrea Richa, and Christian Scheideler. A jamming-resistant mac protocol for single-hop wireless networks. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 45–54. ACM, 2008.

[34] R. Bar-Yehuda, A. Israeli, and A. Itai. Multiple communication in multi-hop radio networks. *SIAM Journal on Computing*, 22(4):875–887, 1993.

[35] Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *Journal of Computer and System Sciences*, 45(1):104–126, 1992.

[36] Bradley J Barnes, Barry Rountree, David K Lowenthal, Jaxk Reeves, Bronis De Supinski, and Martin Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 368–377. ACM, 2008.

[37] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, April 2017.

[38] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-scale in-memory join processing using RDMA. In *International Conference on Management of Data (SIGMOD)*, pages 1463–1475, May 2015.

[39] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44, October 2009.

[40] Rida Bazzi and Gil Neiger. Optimally simulating crash failures in a byzantine environment. In *International Workshop on Distributed Algorithms (WDAG)*, pages 108–128. Springer, 1991.

[41] Jonathan Behrens, Sagar Jha, Matthew Milano, Edward Tremel, Ken Birman, and Robbert van Renesse. The Derecho project. https://derecho-project.github.io.

[42] Jonathan Behrens, Ken Birman, Sagar Jha, Matthew Milano, Edward Tremel, Eugene Bagdasaryan, Theo Gkountouvas, Weijia Song, and Robbert Van Renesse. Derecho: Group communication at the speed of light. Technical report, Technical Report. Cornell University, 2016.

[43] Naama Ben-David and Guy E Blelloch. Analyzing contention and backoff in asynchronous shared memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–62. ACM, 2017.

[44] Naama Ben-David, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–156, 2016.

[45] Naama Ben-David, David Yu Cheng Chan, Vassos Hadzilacos, and Sam Toueg. k-abortable objects: progress under high contention. In *International Symposium on Distributed Computing (DISC)*, pages 298–312. Springer, 2016.

[46] Naama Ben-David, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Implicit decomposition for write-efficient connectivity algorithms. pages 711–722. IEEE, 2018.

[47] Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2019.

[48] Naama Ben-David, Guy E Blelloch, Yihan Sun, and Yuanhao Wei. Multiversion concurrency with bounded delay and precise garbage collection. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 241–252, 2019.

[49] Naama Ben-David, Ziv Scully, and Guy Blelloch. Severus, August 2019. URL https://doi.org/10.5281/zenodo.3360044.

[50] Naama Ben-David, Ziv Scully, and Guy E Blelloch. Unfair scheduling patterns in NUMA architectures. In *International Conference on Parallel Architecture and Compilation (PACT)*, pages 205–218. IEEE, 2019.

[51] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–30, August 1983.

[52] Michael A Bender, Martin Farach-Colton, Simai He, Bradley C Kuszmaul, and Charles E Leiserson. Adversarial contention resolution for simple channels. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 325–332. ACM, 2005.

[53] Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Maxwell Young. How to scale exponential backoff: Constant throughput, polylog access attempts, and robustness. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 636–654. Society for Industrial and Applied Mathematics, 2016.

[54] Michael A Bender, Tsvi Kopelowitz, Seth Pettie, and Maxwell Young. Contention resolution with log-logstar channel accesses. pages 499–508, 2016.

[55] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 168–176, March 1990.

[56] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *International Conference on Principles of Distributed Systems (OPODIS)*, volume 46, 2016.

[57] Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. Sharing memory between byzantine processes using policy-enforced tuple spaces. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):419–432, 2009.

[58] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *USENIX Annual Technical Conference (ATC)*, 2011.

[59] Guy Blelloch, Phillip Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The parallel persistent memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[60] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, and Julian Shun. Efficient algorithms with asymmetric read and write costs. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.

[61] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing paxos. *ACM Sigact News*, 34(1):47–67, 2003.

[62] Sol Boucher, Anuj Kalia, and David G. Andersen. Putting the "micro" back in microservice. pages 645–650, July 2018.

[63] Zohir Bouzid, Damien Imbs, and Michel Raynal. A necessary condition for byzantine k-set agreement. *Information Processing Letters*, 116(12):757–759, 2016.

[64] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[65] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[66] Francisco Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *International Conference on Parallel Computing Technologies*, pages 42–50. Springer, 2001.

[67] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350, 2006.

[68] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 83–97, December 2013.

[69] Irina Calciu, JE Gottschlich, and Maurice Herlihy. Using delegation and elimination to implement a scalable numa-friendly stack. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2013.

[70] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K Aguilera. Black-box concurrent data structures for numa architectures. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 51(2): 207–221, 2017.

[71] Laura Carrington, Allan Snavely, and Nicole Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22(3):336–346, 2006.

[72] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186, February 1999.

[73] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3), August 2003.

[74] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM SIGPLAN Notices*, volume 49, pages 433–452. ACM, 2014.

[75] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs? In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 61–73, October 1994.

[76] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, March 1996.

[77] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, July 1996.

[78] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, August 2007.

[79] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[80] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. Nvmove: Helping programmers move to byte-based persistence. In *INFLOW (OSDI)*, 2016.

[81] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[82] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 189–204, 2007.

[83] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX Annual Technical Conference (ATC)*, pages 287–292, 2008.

[84] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 153–168, 2009.

[85] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (limited) power of non-equivocation. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 301–308. ACM, 2012.

[86] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, 2013.

[87] Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 254–263. ACM, 2015.

[88] Nachshon Cohen, Michal Friedman, and James R Larus. Efficient logging in non-volatile memory by exploiting coherency protocols. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):67, 2017.

[89] Nachshon Cohen, Rachid Guerraoui, and Mihail Igor Zablotchi. The inherent cost of remembering consistently. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[90] Alexei Colin and Brandon Lucia. Termination checking and task decomposition for task-based intermittent programs. In *International Conference on Compiler Construction*, 2018.

[91] Colin Cooper, Tomasz Radzik, Nicolas Rivera, and Takeharu Shiraga. Fast plurality consensus in regular expanders. In *International Symposium on Distributed Computing (DISC)*, pages 13:1–13:16, October 2017.

[92] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 271–282. ACM, 2018.

[93] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 174–183, 2004.

[94] Miguel Correia, Giuliana S Veronese, and Lau Cheuk Lung. Asynchronous byzantine consensus with 2f+ 1 processes. In *Symposium On Applied Computing (SAC)*, pages 475–480. ACM, 2010.

[95] Alexandros Daglis, Dmitrii Ustiugov, Stanko Novakovic, Edouard Bugnion, Babak Falsafi, and Boris Grot. SABRes: Atomic object reads for in-memory rack-scale computing. In *International Symposium on Microarchitecture (MICRO)*, pages 1–13, October 2016.

[96] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. RPCValet: NI-driven tail-aware balancing of μs-scale RPCs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 35–48, April 2019.

[97] Al Danial. cloc: Count lines of code. `https://github.com/AlDanial/cloc`.

[98] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–48. ACM, 2013.

[99] Tudor David, Rachid Guerraoui, and Maysam Yabandeh. Consensus inside. In *International Middleware Conference*, pages 145–156, December 2014.

[100] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *USENIX Annual Technical Conference (ATC)*. 2018.

[101] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent processor architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011.

[102] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, December 2004.

[103] Dan Dobre and Neeraj Suri. One-step consensus with zero-degradation. In *International Conference on Dependable Systems and Networks (DSN)*, pages 137–146. IEEE Computer Society, 2006.

[104] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97, January 1987.

[105] Travis Downs. A benchmark for low-level cpu micro-architectural features. `https://github.com/travisdowns/uarch-bench`.

[106] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, April 2014.

[107] Aleksandar Dragojevic, Dushyanth Narayanan, Ed Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2015.

[108] Partha Dutta, Rachid Guerraoui, and Leslie Lamport. How fast can eventual synchrony lead to consensus? In *International Conference on Dependable Systems and Networks (DSN)*, pages 22–27. IEEE, 2005.

[109] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, pages 288–323, April 1988.

[110] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM (JACM)*, 44(6):779–805, 1997.

[111] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. Snzi: Scalable nonzero indicators. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 13–22. ACM, 2007.

[112] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140. ACM, 2010.

[113] Faith Ellen, Danny Hendler, and Nir Shavit. On the inherent sequentiality of concurrent objects. *SIAM Journal on Computing*, 41(3):519–536, 2012.

[114] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2014.

[115] Faith Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 2003.

[116] Faith Ellen Fich, Danny Hendler, and Nir Shavit. Linear lower bounds on real-world implementations of concurrent objects. pages 165–173. IEEE, 2005.

[117] Jeremy T Fineman, Seth Gilbert, Fabian Kuhn, and Calvin Newport. Contention resolution on a fading channel. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 155–164. ACM, 2016.

[118] Jeremy T Fineman, Calvin Newport, and Tonghe Wang. Contention resolution on multiple channels with collision detection. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 175–184. ACM, 2016.

[119] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, April 1985.

[120] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6):1–40, 2011.

[121] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.

[122] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 28–40. ACM, 2018.

[123] 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. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[124] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[125] Ofer Gabber and Zvi Galil. Explicit constructions of linear-sized superconcentrators. *Journal of Computer and System Sciences*, 22(3):407–420, June 1981.

[126] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[127] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–18, April 2019.

[128] genz. Gen-Z draft core specification—december 2016. `http://genzconsortium.org/draft-core-specification-december-2016`.

[129] Mohsen Ghaffari, Bernhard Haeupler, and Majid Khabbazian. Randomized broadcast in radio networks with collision detection. *Distributed Computing*, 28(6):407–422, 2015.

[130] Phillip B Gibbons, Yossi Matias, and Vijaya Ramachandran. Efficient low-contention parallel algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 236–247. ACM, 1994.

[131] Phillip B Gibbons, Yossi Matias, and Vijaya Ramachandran. The queue-read queue-write asynchronous PRAM model. In *International Conference on Parallel Processing (EuroPar)*, pages 277–292. Springer, 1996.

[132] Phillip B Gibbons, Yossi Matias, and Vijaya Ramachandran. The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, pages 638–648, 1997.

[133] Wojciech Golab and Danny Hendler. Recoverable mutual exclusion under system-wide failures. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 17–26, 2018.

[134] Daniel Goodman, Georgios Varisteas, and Timothy L. Harris. Pandia: comprehensive contention-sensitive thread placement. In *European Conference on Computer Systems (EuroSys)*, pages 254–269. ACM, 2017.

[135] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, 1990.

[136] Yan Gu. *Write-efficient Algorithms*. PhD thesis, Intel, 2018.

[137] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *ACM Conference on SIGCOMM*, pages 202–215, August 2016.

[138] Daniel Hackenberg, Daniel Molka, and Wolfgang E Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. In *International Symposium on Microarchitecture (MICRO)*, pages 413–422. IEEE, 2009.

[139] Vassos Hadzilacos, Xing Hu, and Sam Toueg. Optimal register construction in m&m systems. *International Conference on Principles of Distributed Systems (OPODIS)*, 2019.

[140] Bernhard Haeupler and David Wajc. A faster distributed radio broadcast primitive. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[141] Syed Kamran Haider, William Hasenplaugh, and Dan Alistarh. Lease/release: Architectural support for scaling contended data structures. *ACM SIGPLAN Notices*, 51(8):17, 2016.

[142] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Workshop on Hot Topics in Networks (HOTNETS)*, pages 10:1–10:7, November 2013.

[143] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[144] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing (DISC)*, pages 300–314. Springer, 2001.

[145] Timothy L. Harris. Five ways not to fool yourself: designing experiments for understanding performance. `https://timharris.uk/misc/five-ways.pdf`, 2016.

[146] Danny Hendler and Shay Kutten. Constructing shared objects that are both robust and high-throughput. In *International Symposium on Distributed Computing (DISC)*, pages 428–442. Springer, 2006.

[147] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364. ACM, 2010.

[148] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *ACM SIGPLAN Notices*, volume 25, pages 197–206. ACM, 1990.

[149] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[150] Maurice Herlihy and Nir Shavit. On the nature of progress. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 313–328. Springer, 2011.

[151] Maurice Herlihy, Nir Shavit, and Orli Waarts. Linearizable counting networks. *Distributed Computing*, 9(4):193–203, 1996.

[152] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 2003.

[153] Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed computing through combinatorial topology*. Morgan Kaufmann, 2013.

[154] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, January 1990.

[155] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[156] Brandon Holt, James Bornholt, Irene Zhang, Dan R. K. Ports, Mark Oskin, and Luis Ceze. Disciplined inconsistency with consistency types. In *Symposium on Cloud Computing*

(SoCC), pages 279–293, 2016.

[157] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, August 2006.

[158] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, June 2010.

[159] Shams Mahmood Imam and Vivek Sarkay. Habanero-java library: a java 8 framework for multicore programming. In *International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ*, 2014.

[160] Infiniband. InfiniBand. `http://www.infinibandta.org/content/pages.php?pg=about_us_infiniband`.

[161] Intel. Intel threading building blocks, 2011. `https://www.threadingbuildingblocks.org/`.

[162] Intel. Intel64 and ia-32 architectures optimization reference manual, 2016. URL `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`.

[163] IntelOmnipath. Intel Omni-Path. `http://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html`.

[164] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 425–438, 2016.

[165] iwarp. iWARP. `https://en.wikipedia.org/wiki/IWARP`.

[166] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing (DISC)*, pages 313–327. Springer, 2016.

[167] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM (JACM)*, 45(3):451–500, May 1998.

[168] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.

[169] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 35–49, 2018.

[170] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *ACM Conference on SIGCOMM*, pages 295–306, August 2014.

[171] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. *ACM SIGCOMM Computer Communication Review*, 44(4):295–306, 2015.

[172] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high perfor-

mance RDMA systems. In *USENIX Annual Technical Conference (ATC)*, pages 437–450, 2016.

[173] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 185–201, November 2016.

[174] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–16, February 2019.

[175] Anuj Kalia Michael Kaminsky and David G Andersen. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conference (ATC)*, page 437, 2016.

[176] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: resource-efficient byzantine fault tolerance. In *European Conference on Computer Systems (EuroSys)*, pages 295–308, 2012.

[177] Richard M Karp and Yanjun Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM (JACM)*, 40(3):765–789, 1993.

[178] Antonios Katsarakis, Vasilis Avrielatos, M R Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojević, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 201–217, March 2020.

[179] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 3–14, June 2015.

[180] 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.

[181] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *ACM SIGPLAN international conference on Functional Programming (ICFP)*, 2010.

[182] Darren J Kerbyson, Henry J Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J Wasserman, and Mike Gittings. Predictive performance and scalability modeling of a large-scale application. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 39–39. IEEE, 2001.

[183] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 297–312, 2018.

[184] A. C. Klaiber and H. M. Levy. A comparison of message passing and shared memory

architectures for data parallel programs. In *International Symposium on Computer Architecture (ISCA)*, pages 94–105, April 1994.

[185] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[186] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[187] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *European Conference on Computer Systems (EuroSys)*, April 2020.

[188] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. *ACM SIGOPS Operating Systems Review*, 50(2):399–411, 2016.

[189] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, October 2007.

[190] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message-passing and shared-memory: Early experience. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 54–63, 1993.

[191] Klaus Kursawe. Optimistic byzantine agreement. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 262–267, October 2002.

[192] Leslie Lamport. The weak byzantine generals problem. *Journal of the ACM (JACM)*, 30 (3):668–676, July 1983.

[193] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[194] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[195] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[196] Leslie Lamport. Fast paxos. *Distributed Computing*, pages 79–103, October 2006.

[197] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[198] Butler W Lampson. How to build a highly available system using consensus. In *International Workshop on Distributed Algorithms (WDAG)*, pages 1–17, 1996.

[199] Doug Lea. A java fork/join framework. In *Proceedings of the ACM Conference on Java Grande*, JAVA '00, 2000.

[200] T.J. LeBlanc and Evangelos Markatos. Shared memory vs. message passing in shared-memory multiprocessors. In *IEEE Symposium on Parallel and Distributed Processing*, pages 254 – 263, December 1992.

[201] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. Wort:

Write optimal radix tree for persistent memory storage systems. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 257–270, 2017.

[202] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: converting concurrent dram indexes to persistent-memory indexes. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 462–477, 2019.

[203] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[204] Herwig Lejsek, Friðrik Heiðar Ásmundsson, Björn Þór Jónsson, and Laurent Amsaleg. Nv-tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):869–883, 2009.

[205] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the FALCON spy network. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2011.

[206] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.

[207] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed systems with help from the network. In *European Conference on Computer Systems (EuroSys)*, April 2015.

[208] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: datacenter sockets can be fast and compatible. In *ACM Conference on SIGCOMM*, pages 90–103, August 2019.

[209] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 467–483, 2016.

[210] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy M Lohman. Numa-aware algorithms: the case of data shuffling. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[211] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *International Symposium on Computer Architecture (ISCA)*, pages 267–278, June 2009.

[212] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 189–200, February 2012.

[213] liquibook. Liquibook. `https://github.com/enewhuis/liquibook`. Accessed 2020-05-25.

[214] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32 (3):167–198, 2004. doi: 10.1023/B:IJPP.0000029272.69895.c1. URL https://doi.org/10.1023/B:IJPP.0000029272.69895.c1.

[215] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–343. ACM, 2017.

[216] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartNICs using iPipe. In *ACM Conference on SIGCOMM*, pages 318–333, August 2019.

[217] Pangfeng Liu, William Aiello, and Sandeep Bhatt. An atomic model for message-passing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 154–163. ACM, 1993.

[218] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael Scott, Sam H. Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *International Symposium on Microarchitecture (MICRO)*, pages 258–270, 2018.

[219] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *International Symposium on Microarchitecture (MICRO)*, pages 258–270. IEEE, 2018.

[220] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[221] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *USENIX Annual Technical Conference (ATC)*, pages 773–785, July 2017.

[222] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. *PLDI*, 2015.

[223] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[224] Dahlia Malkhi, Michael Merritt, Michael K Reiter, and Gadi Taubenfeld. Objects shared by byzantine processes. *Distributed computing (DIST)*, 16(1):37–48, 2003.

[225] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 3(3):202–215, July 2006.

[226] David Mazieres. Paxos made practical. https://www.scs.stanford.edu/~dm/home/papers/paxos.pdf, 2007.

[227] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9 (1):21–65, 1991.

[228] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *European Conference on Computer Systems (EuroSys)*,

pages 499–512. ACM, 2017.

[229] memcached. Memcached. `https://memcached.org/`. Accessed 2020-05-25.

[230] Robert M Metcalfe and David R Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[231] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel & Distributed Systems*, (6):491–504, 2004.

[232] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275. ACM, 1996.

[233] Maged M Michael and Michael L Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *journal of parallel and distributed computing*, 51(1):1–26, 1998.

[234] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX Annual Technical Conference (ATC)*, pages 103–114, June 2013.

[235] Mark Moir and Nir Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 47–14, 2007.

[236] Daniel Molka, Daniel Hackenberg, and Robert Schöne. Main memory and cache performance of intel sandy bridge and amd bulldozer. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, page 4. ACM, 2014.

[237] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B Morrey III, Dhruva R Chakrabarti, and Michael L Scott. Dalí: A periodically persistent hash map. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 91. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[238] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[239] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX Annual Technical Conference (ATC)*, pages 291–305, July 2015.

[240] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–18, March 2014.

[241] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. Storm: a fast transactional dataplane for remote data structures. In *ACM International Conference on Systems and Storage (SYSTOR)*, pages 97–108, May 2019.

[242] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, pages 305–320, June 2014.

[243] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosen-

blum. Fast crash recovery in RAMCloud. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–41, October 2011.

[244] Rotem Oshman and Nir Shavit. The skiptrie: Low-depth concurrent search without rebalancing. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2013.

[245] 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 *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386. ACM, 2016.

[246] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 361–378, February 2019.

[247] Seo Jin Park and John Ousterhout. Exploiting commutativity for practical fast replication. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 47–64, February 2019.

[248] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[249] Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, April 2002.

[250] Marius Poke and Torsten Hoefler. DARE: High-performance state machine replication on RDMA networks. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 107–118, June 2015.

[251] Mary C. Potter, Brad Wyble, Carl Erick Hagmann, and Emily Sarah McCourt. Detecting meaning in RSVP at 13 ms per picture. *Attention, Perception, & Psychophysics*, 76(2): 270–279, 2014.

[252] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Symposium on Cloud Computing (SoCC)*, pages 342–355, August 2015.

[253] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, October 2017.

[254] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 145–160, October 2018.

[255] rdmaroce. RDMA over converged ethernet. `https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet`.

[256] redis. Redis. `https://redis.io/`. Accessed 2020-05-25.

[257] Tim Roughgarden. Beyond the worst-case analysis of algorithms, 2020.

[258] Signe Rüsch, Ines Messadi, and Rüdiger Kapitza. Towards low-latency byzantine agree-

ment protocols using RDMA. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 146–151. IEEE, 2018.

[259] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 174–185, October 1996.

[260] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2005.

[261] William N Scherer III and Michael L Scott. Advanced contention management for dynamic software transactional memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 240–248. ACM, 2005.

[262] David Schneider. The microsecond market. *IEEE Spectrum*, 49(6), June 2012.

[263] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[264] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures. In *International Conference on Parallel Architecture and Compilation (PACT)*, pages 445–456. IEEE, 2015.

[265] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 386–400, October 2011.

[266] Nir Shavit and Dan Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1995.

[267] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 317–332, November 2016.

[268] SNIA. Extending RDMA for persistent memory over fabrics. `https://www.snia.org/sites/default/files/ESF/Extending-RDMA-for-Persistent-Memory-over-Fabrics-Final.pdf`.

[269] Wonjun Song, Gwangsun Kim, Hyungjoon Jung, Jongwook Chung, Jung Ho Ahn, Jae W Lee, and John Kim. History-based arbitration for fairness in processor-interconnect of numa servers. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 51(2):765–777, 2017.

[270] Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In *International Symposium on Distributed Computing (DISC)*, pages 438–450, September 2008.

[271] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. DaRPC: Data center RPC. In *Symposium on Cloud Computing (SoCC)*, pages 1–13, November 2014.

[272] R Michael Tanner. Explicit concentrators from generalized N-gons. *SIAM Journal on Algebraic Discrete Methods*, 5(3):287–293, 1984.

[273] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014.

[274] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 49, pages 357–368. ACM, 2014.

[275] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 306–324, October 2017.

[276] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *USENIX Conference on File and Storage Technologies (FAST)*, volume 11, pages 61–75, 2011.

[277] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1): 16–30, 2013.

[278] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 39, pages 91–104. ACM, 2011.

[279] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: Fast and scalable PAXOS on RDMA. In *Symposium on Cloud Computing (SoCC)*, pages 94–107, September 2017.

[280] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–104, 2015.

[281] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 233–251, 2018.

[282] Udara Wijetunge, Sylvie Perreau, and Andre Pollok. Distributed stochastic routing optimization using expander graph theory. In *Australian Communications Theory Workshop*, pages 124–129, January 2011.

[283] Timothy S. Woodall, Galen M. Shipman, George Bosilca, Richard L. Graham, and Arthur B. Maccabe. High performance RDMA protocols in HPC. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting*, pages 76–85, September 2006.

[284] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. The end of a myth: Distributed transactions can scale. *Proceedings of the VLDB Endowment*, 10(6):685–696, February 2017.

[285] Jidong Zhai, Wenguang Chen, and Weimin Zheng. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. In *ACM Sigplan*

Notices, volume 45, pages 305–314. ACM, 2010.