THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e. g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

TANDEM: Taming Failures In Next-Generation Datacenters With Emerging Memory

Mahesh Dananjaya

Doctor of Philosophy

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2023

Abstract

The explosive growth of online services, leading to unforeseen scales, has made modern datacenters highly prone to failures. Taming these failures hinges on fast and correct recovery, minimizing service interruptions. Applications, owing to recovery, entail additional measures to maintain a recoverable state of data and computation logic during their failure-free execution. However, these precautionary measures have severe implications on performance, correctness, and programmability, making recovery incredibly challenging to realize in practice.

Emerging memory, particularly non-volatile memory (NVM) and disaggregated memory (DM), offers a promising opportunity to achieve fast recovery with maximum performance. However, incorporating these technologies into datacenter architecture presents significant challenges; Their distinct architectural attributes, differing significantly from traditional memory devices, introduce new semantic challenges for implementing recovery, complicating correctness and programmability. Can emerging memory enable fast, performant, and correct recovery in the datacenter? This thesis aims to answer this question while addressing the associated challenges.

When architecting datacenters with emerging memory, system architects face four key challenges: ① how to guarantee correct semantics; ② how to efficiently enforce correctness with optimal performance; ③ how to validate end-to-end correctness including recovery; and ④ how to preserve programmer productivity (Programmability).

This thesis aims to address these challenges through the following approaches: ⓐ defining precise consistency models that formally specify correct end-to-end semantics in the presence of failures (consistency models also play a crucial role in programmability); ⓑ developing new low-level mechanisms to efficiently enforce the prescribed models given the capabilities of emerging memory; and ⓒ creating robust testing frameworks to validate end-to-end correctness and recovery.

We start our exploration with non-volatile memory (NVM), which offers fast persistence capabilities directly accessible through the processor's load-store (memory) interface. Notably, these capabilities can be leveraged to enable fast recovery for Log-Free Data Structures (LFDs) while maximizing performance. However, due to the complexity of modern cache hierarchies, data hardly persist in any specific order, jeopardizing recovery and correctness. Therefore, recovery needs primitives that explicitly control the order of updates to NVM (known as persistency models). We outline the precise specification of a novel persistency model – Release Persistency (RP) – that provides a consistency guarantee for LFDs on what remains in non-volatile memory upon

failure. To efficiently enforce RP, we propose a novel microarchitecture mechanism, lazy release persistence (LRP). Using standard LFDs benchmarks, we show that LRP achieves fast recovery while incurring minimal overhead on performance.

We continue our discussion with memory disaggregation which decouples memory from traditional monolithic servers, offering a promising pathway for achieving very high availability in replicated in-memory data stores. Achieving such availability hinges on transaction protocols that can efficiently handle recovery in this setting, where compute and memory are independent. However, there is a challenge: disaggregated memory (DM) fails to work with RPC-style protocols, mandating one-sided transaction protocols. Exacerbating the problem, one-sided transactions expose critical low-level ordering to architects, posing a threat to correctness. We present a highly available transaction protocol, Pandora, that is specifically designed to achieve fast recovery in disaggregated key-value stores (DKVSes). Pandora is the first one-sided transactional protocol that ensures correct, non-blocking, and fast recovery in DKVS. Our experimental implementation artifacts demonstrate that Pandora achieves fast recovery and high availability while causing minimal disruption to services.

Finally, we introduce a novel target litmus-testing framework – DART – to validate the end-to-end correctness of transactional protocols with recovery. Using DART's target testing capabilities, we have found several critical bugs in Pandora, highlighting the need for robust end-to-end testing methods in the design loop to iteratively fix correctness bugs. Crucially, DART is lightweight and black-box, thereby eliminating any intervention from the programmers.

Lay summary

Modern datacenters have continued to grow in size and complexity, with failures becoming more common than exceptions. The frequency of these failures and their complexity and impact have dramatically escalated in recent years. These failures, unfortunately, disrupt critical services and reduce the datacenters' ability to respond to users online (i.e., availability). Such disruptions not only incur significant costs but also tarnish the reputation of cloud operators.

Addressing failures requires fast recovery algorithms to restore components and avoid service interruptions. However, achieving fast recovery involves trade-offs in performance, correctness, and programmability. Emerging non-volatile and disaggregated memories promise opportunities for efficient recovery but also introduce correctness and programmability challenges due to their unique architectures.

In addressing these challenges, this thesis tackles three key aspects of architecting modern datacenters with emerging memory: defining precise correctness semantics (i.e., consistency), developing performant low-level primitives for efficiently enforcing correctness, and creating robust testing frameworks to validate end-to-end correctness.

This thesis makes significant contributions in three key areas: Firstly, we propose a novel consistency model, Release Persistency, along with an efficient microarchitecture mechanism, Lazy Release Persistency. These innovations are designed to enable fast, correct, and performant recovery in non-volatile-memory data structures. Secondly, leveraging the benefits of disaggregated memory, we introduce Pandora, the first fast, correct, and recoverable one-sided transaction protocol specifically designed for disaggregated data stores. Finally, we introduce DART, a framework that validates end-to-end correctness of transaction protocols with recovery.

Overall, this thesis exemplifies a holistic approach addressing recovery (availability), performance, consistency, and programmability tensions using emerging memories. The specifications, mechanisms, and techniques unlock potential for reliable, efficient recovery, avoiding catastrophic failures in next-generation datacenters. Despite limitations, it provides promising directions for continued research on next-generation datacenter architectures.

Acknowledgements

This PhD thesis would not have been possible without the support of many individuals.

First and foremost, I express my deepest gratitude to my supervisors, Vijay Nagarajan, Murray Cole, Pramod Bhatotia, and Vaishak Belle. Their invaluable guidance, mentorship and unwavering belief in my abilities have been instrumental throughout my PhD journey. I am immensely grateful for the knowledge and wisdom they have imparted. I also wish to extend profound thanks to my thesis committee members, Yuvraj Patel and Haris Volos, for their constructive feedback and steadfast support in shaping this thesis.

I extend special thanks to Vasilis Gavrielatos for being an amazing collaborator and mentor from day one. His support and guidance throughout my PhD was pivotal for its successful completion. I am also thankful to my other collaborators and mentors, including Arpit Joshi, Antonis Katsarakis, Rodrigo Rocha, Sukarn Agarwal, and Andres Goens, for their support and encouragement throughout this journey.

I am thankful to my academic colleagues and friends, whose camaraderie, intellectual exchanges, and shared passion for research have made this academic pursuit both challenging and enjoyable. Their diverse perspectives and stimulating discussions have enriched my understanding and inspired new ideas. I cherish the friendships formed during this time and appreciate the collaborative spirit that permeates our academic community. (There are so many to mention, this is for all of you.)

I am indebted to my family and Tharu for their unconditional love, unwavering support, and constant encouragement. Their belief in my dreams and their sacrifices have been a constant source of strength and motivation. Their presence in my life has been a beacon of light during the most challenging times, and I am profoundly grateful for their unwavering faith in me. Additionally, I am blessed to have found an extended family in Edinburgh — Pete, Matt, Rado, Muyang, Roger, Roberta, and all my CDT friends. They have always been a constant source of inspiration to me.

Finally, I dedicate this thesis to my beloved mother, whose love, guidance, and sacrifices have been the foundation of my life's journey. Her unwavering belief in my potential, even during moments of self-doubt, has instilled in me the courage to pursue my dreams relentlessly. Her love and encouragement have been a driving force behind my academic achievements, and I am forever grateful for her presence in my life. This thesis is a testament to her unwavering support and a humble token of my love and appreciation.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

- M. Dananjaya, “ Release Persistency ”, Master of Science by Research Thesis, The University of Edinburgh.
- M. Dananjaya, V. Gavrielatos, A. Joshi, V. Nagarajan, “ Lazy Release Persistency ”, 2020 ACM 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020, Lausanne, Switzerland. doi: 10.1145/3373376.3378481.

(*Mahesh Dananjaya*)

To my loving mother for instilling the value of consistency, persistency, and taming failures in my life

Contents

1 Introduction **1**

1.1 Motivation . . . 2

1.1.1 Failures and Availability . . . 2

1.1.2 Recovery . . . 3

1.1.3 Recovery Trade-Offs . . . 3

1.2 Impact of Emerging Memory . . . 5

1.2.1 Non-Volatile Memory (NVM) . . . 5

1.2.2 Disaggregated Memory (DM) . . . 6

1.3 This Thesis . . . 8

1.3.1 Problem Statement . . . 8

1.3.2 Research Objectives . . . 10

1.3.3 Scope . . . 11

1.4 Thesis Contributions . . . 11

1.4.1 Lazy Release Persistency (LRP) . . . 11

1.4.2 Pandora . . . 12

1.4.3 DART . . . 12

1.5 Summary . . . 13

2 Background **15**

2.1 The Datacenter Architecture . . . 15

2.1.1 Recovery . . . 16

2.2 Emerging Memory Systems . . . 18

2.2.1 Non-Volatile Memory (NVM) . . . 18

2.2.2 Disaggregated Memory (DM) . . . 19

2.3 Consistency Models . . . 20

2.3.1 Linearizability . . . 21

2.3.2 Sequential Consistency . . . 21

2.3.3 Release Consistency 22
2.4 Memory Persistency Models 23
2.5 Transactional Consistency Models 24
2.5.1 Strict Serializability 25

3 Lazy Release Persistency 29
3.1 Introduction 29
3.1.1 Contributions 31
3.2 Preliminaries 32
3.2.1 Release Consistency 32
3.2.2 Persistency Models 33
3.2.3 Log-free data structures (LFDs) 34
3.3 Limitations of ARP 35
3.3.1 ARP semantics 36
3.3.2 ARP implementation 36
3.3.3 Why not simply fix ARP? 37
3.4 Release Persistency 38
3.4.1 Formal Specification 38
3.4.2 Specification implications 39
3.5 Lazy Release Persistency 40
3.5.1 LRP Overview 40
3.5.2 LRP: Microarchitecture 42
3.6 Experimental Evaluation 47
3.6.1 Workloads 47
3.6.2 Comparison Points 48
3.6.3 Simulator 49
3.6.4 Results 50
3.7 Summary 53

4 Pandora: Highly Available, Recoverable Transactions on Disaggregated Data Stores 57
4.1 Introduction 57
4.1.1 Pandora 59
4.1.2 Contributions 60
4.2 Preliminaries 61
4.2.1 Disaggregated KVS (DKVS) 61

4.2.2 Recoverable Transaction Protocol 63
4.2.3 FORD 64
4.3 Pandora 65
4.3.1 Making FORD Efficiently Recoverable 65
4.3.2 Recovery Protocol 69
4.4 Evaluating Pandora: Goals and Methodology 72
4.4.1 Methodology 73
4.5 Experimental Evaluation 74
4.5.1 Recovery Latency 74
4.5.2 Steady-State Throughput 76
4.5.3 Fail-Over Throughput 79
4.6 Summary 82

5 DART: Validating Transactional Databases with Target Application-Centric Recurrent Testing 85
5.1 Introduction 85
5.2 Method 86
5.2.1 Application-Centric Assertions 87
5.3 Litmus Tests, Bugs, and Fixes 88
5.3.1 Litmus 1 88
5.3.2 Litmus 2 89
5.3.3 Litmus 3 90
5.3.4 More Bugs and Fixes 90
5.4 Test Coverage 91
5.5 Summary 92

6 Conclusion 95
6.1 Summary 95
6.2 Critical Analysis 97
6.3 Lessons Learned 99
6.3.1 Lesson 1: Formal Methods vs Testing 99
6.3.2 Lesson 2: Inherent Complexity of Consistency 100
6.4 Future Work 101
6.4.1 Formal Proof 101
6.4.2 Verifiably Correct Transaction Protocols 101
6.4.3 Incorporating New Emerging Memory Technologies 102

6.4.4 Final Remarks 102

Bibliography **103**

Chapter 1

Introduction

From shopping to socializing, our everyday activities are increasingly reliant on a wide range of online services, including e-commerce, self-driving, banking, streaming, social media, and online gaming. These services run in a shim layer, known as Cloud [19, 20] or Sky [196], functioning over datacenters.

The unprecedented growth of online services has led to previously unseen demand for datacenters. To meet such a proliferating demand, modern data centers have become increasingly large, complex, and heterogeneous, making failures the norm rather than the exception. Unfortunately, failure disrupts critical services implemented in modern datacenters [27, 85].

The cost of service downtime is substantial, resulting in significant financial losses for businesses and negatively impacting overall quality of service (QoS). A typical service outage of a few minutes can cost millions of dollars for companies [125]. Therefore, it is critically important to architect data centers with fault tolerance [71, 57].

Tolerating failures in modern data centers requires fast recovery to minimize interruptions to the services (i.e., high availability). However, fast recovery gives rise to new challenges in performance, correctness, and programmability.

In this chapter, we closely examine failures, recovery, and their crucial trade-offs in the data center while outlining the problem statement, research objectives, and the scope of this thesis (§1.1). Furthermore, we describe our approach and the contributions of this thesis (§1.4).

1.1 Motivation

The modern datacenter has created an abstract machine–which is referred to as the Cloud, or Sky [19, 20, 196]– to store, retrieve, and process petabytes of application data. Such abstraction allows for seamless management of online services while hiding the underlying scale and complexity from application developers. However, the reality is that modern datacenters consist of countless numbers of hardware and software resources (CPU, memory, storage, network and OS libraries, application software) that fail all the time [109, 27]. In this section, we discuss the importance of taming failures in modern datacenters.

1.1.1 Failures and Availability

In 2013, a study conducted by Google revealed that a typical datacenter server experiences 1.2 to 2 crashes per year [26]. Even at these rates, the mean time to failure (MTTF) of the datacenter, comprising tens of thousands of servers, was estimated to be in the order of minutes. For instance, consider a datacenter with 10,000 servers, each having an MTTF of 365 days (equivalent to one crash per year). The calculated MTTF for this datacenter would be approximately 50 minutes, which is significantly low (calculated as $365 days * 24 hrs/day * 60 mins/hr * 1/10,000 = 50 mins$). The today's reality, however, is considerably more challenging, as modern datacenters have continued to scale up to tens of thousands to millions of servers, and the complexity of failures has substantially increased [26, 24, 149].

Failures can disrupt a service in two ways. First, the failed servers stop responding to service requests until the damage is recovered. Second, a failed server can abruptly disrupt the functioning of the remaining servers in the service. Availability in the presence of such failures is measured as a fraction of time that the service remains operational [102]. For instance, consider a service operating on a datacenter with mean time to failures (MTTF) of 60 min and mean time to recover (MTTR) of 500 ms. The calculated availability is 0.99 or two nines (using equation 1.1). Crucially, datacenter operators prioritize offering high availability, ideally five nines and more [49], as it directly affects the client's online experience and profits.

Additionally, failures are not the sole factor affecting availability; high client traffic and limited resource availability can adversely impact overall availability. While these aspects are important, they are not the focus of this thesis.

$$Availability = \frac{MTTF}{MTTF + MTTR} \tag{1.1}$$

1.1.2 Recovery

A typical service involves multiple applications that are implemented over numerous servers, each responsible for computation and storage functionalities. Clients interact with the services by sending requests to these servers. Unfortunately, crashed servers lose data, stop responding to clients, and disrupt the operation of other servers. Maintaining service availability in the face of failures requires quickly and correctly recovering failed segment of the application.

System architects typically use different terms such as recovery and fault tolerance interchangeably to describe a system's ability to operate through failures. While their textbook definitions have subtle nuances, the lines between these concepts have increasingly blurred in real-world usage, largely owing to intricate modern datacenter designs and complex application logic [38]. Therefore, before delving into recovery, let's first explore what it entails.

Traditionally, recovery refers to steps taken after a failure to bring back the failed components to life. However, modern datacenters demand high availability, making it no longer feasible to wait for all the failed components to be fully recovered. Thus, applications incorporate redundancy mechanisms like replication to minimize downtime by relaying the application over redundant resources [38, 198, 202]; clients experience no interruptions. While minimal, these redundancy mechanisms still require a form of recovery which can be as little as resending client requests to a redundant application server [38]. In this thesis, we categorize all these different aspects as recovery.

Broadly, we define recovery as the ability to restore a failed segment of the application – data and computation logic – regardless of whether the application remains on the same physical devices. This definition enables us to reason about recovery in a system-agnostic way.

1.1.3 Recovery Trade-Offs

Given the unpredictable nature of failures, which can occur at any point during application execution, recovery requires implementing additional measures to maintain a recoverable state throughout the failure-free execution of applications.

First, it is vital to store application data in a fail-safe manner, as there is a risk of permanent loss in the event of failures. System architects must ensure that data remains stored and accessible over time, even in the face of various failures or disasters (durability). Durability requires persistence or replicating of data [83, 96, 76]. Persistence ensures that failed servers can restart with a consistent data state, while replication allows the failed portion of the application to continue functioning on a backup server [39, 14].

Second, the consequences of server failures extend beyond data losses to disrupting computation logic. Recovery must ensure that each request appears to execute exactly once in an all-or-nothing manner with no partial effects on data (*Failure Atomicity*). This is because, typically, client requests trigger multiple actions within application servers, hence failures can unpredictably disrupt the execution at any point. In particular, system architects ensure atomicity through write-ahead logging [160, 159, 180], a technique that involves preemptively saving copies of data before making actual changes.

Crucially, additional recovery measures to implement data durability and failure atomicity have severe implications on performance, correctness, and programmability.

Correctness Implications. Recovery complicates the end-to-end correctness of applications. Additional recovery measures introduce significant complexity into the protocols and implementation, making it incredibly difficult to validate against the abstract correctness properties of the applications [98, 197, 46, 142]. Rare corner cases and the non-deterministic nature of failures further exacerbate this problem. Designing recovery protocols and validating their correctness remains a notoriously challenging problem [151, 103, 173, 214, 213].

Performance Implications. Implementing precautionary measures for recovery negatively impacts performance; Since datacenter performance is measured by the number of client requests completed within a fixed time frame (also referred to as throughput), introducing more recovery measures results in higher CPU usage for processing client requests, thereby reducing overall performance.

Programmability Implications. Alternatively, if system architects choose to expose low-level complexities like handling failures to programmers for performance reasons [179, 136, 78], the subtle complexities in recovery can diminish programmer productivity (programmability), and usability in the datacenter.

Design Goals. Achieving fast and correct recovery, while critically important, is a significantly challenging task. Architects focus on four key aspects when designing

datacenter recovery:(i) minimizing recovery time or downtime (High Availability), (ii) reducing performance overhead to a minimum (High Performance), (iii) enforcing end-to-end correctness as if no failures occurred (Consistency), and (iv) preserving programmer productivity (Programmability). However, achieving these objectives remains challenging despite years of research, mainly due to their inherent conflicts and ever-increasing failures. State-of-the-art recovery methods often prove slow, incorrect, or require significant correctness and programmability trade-offs [211, 66].

1.2 Impact of Emerging Memory

Two key innovations in the memory subsystem (known as emerging memory) – non-volatile memory (NVM) and hardware-disaggregated memory (we will simply refer to it as *disaggregated memory* or *DM* from here on) – have radically changed the way in which datacenters are built and operated. Crucially, we observe that these new memory technologies offer a promising pathway for enabling fast recovery with minimal performance overhead. In this section, we provide an intuitive overview of this opportunity. First, we start our discussion with non-volatile memory (NVM) and then continue it with disaggregated memory.

1.2.1 Non-Volatile Memory (NVM)

Datacenter applications largely rely on memory for low latency and high performance. Historically, the content of this memory was lost when servers failed by crashing (volatile). In contrast, the content of NVM survives server failures, hence offering persistent storage. What does this mean for recovery?

In the case of volatile memory, the application must constantly backup data in persistent storage for durability. However, traditional persistent storage devices such as hard disks, accessible only through the I/O interface, introduce significant latency, typically ranging from hundreds of microseconds to tens of milliseconds, thereby inducing prohibitively large overhead on performance [210]. On the contrary, nonvolatile memory (NVM) provides fast persistence within a few hundred nanoseconds, significantly faster than traditional disk-based storage [112]. This fast persistence capability of NVM can enable recovery with minimal performance overhead.

Opportunity. In-memory data structures are pivotal components in modern datacenters, facilitating ultra-fast storage and retrieval of client and management data for appli-

cations [43]. However, these data structures lack persistence, which means that their contents can be vulnerable to loss in the event of failure. To mitigate the risk of data loss, conventional approaches involve persisting data on slower storage media, such as hard disks. Unfortunately, this method often leads to significantly reduced throughput [215].

Non-volatile memory (NVM) presents an alternative solution for in-memory data structures by eliminating the need for slow persistent storage mediums like hard disks (as memory itself is inherently persistent). This offers the opportunity to achieve maximum performance for applications relying on in-memory data structures [43].

Log-free data structures (LFDs) are a special class of in-memory data structures that can recover fast in the presence of failures (Null Recovery) [111]. This is achieved by preserving their happens-before order, in which concurrent operations are visible, in persistent storage without incorporating additional techniques such as logging [56, 111]. However, LFDs often suffer from high overhead on performance for maintaining a recoverable state during failure-free execution (because they have to backup data on a disk in the critical path of execution). Thus, LFDs coupled with NVM can offer fast recovery with maximum performance [56].

1.2.2 Disaggregated Memory (DM)

Memory disaggregation decouples memory from traditional monolithic servers [144, 145]. In this new setting, *compute servers* execute the application logic while *memory servers* store the application data. Direct communication between compute and memory servers is facilitated by fast Remote Direct Memory Access (RDMA) [119], or future technologies such as Compute eXpress Link (CXL) [139, 90].

While disaggregation is not a novel concept and has been previously applied to various components within datacenters, such as storage [25, 84, 155], power [172], and cooling systems [171], the advent of fully-fledged hardware-disaggregated memory has rekindled the interest of system architects for several compelling reasons. First, memory, despite its cost [139], often remains underutilized due to over-provisioning based on peak load estimations. Second, the demand for memory is experiencing exponential growth as big data processing increasingly relies on memory for faster execution. However, the capacity of individual server memory is limited by vendor-specific interconnection technologies and other hardware constraints [9]. This trend is driving datacenters toward resembling distributed shared memory systems (DSMs) [59, 120]. Unlike traditional DSMs that heavily rely on Remote Procedure Calls (RPCs), which

are no longer viable in next-generation data centers [177, 216], memory disaggregation provides a way to tightly couple remote memory with application logic.

In addition, DM offers significant advantages in terms of cost savings, energy efficiency, scalability, and manageability in the datacenter [99, 130].

Opportunity. What does memory disaggregation mean to recovery?

Disaggregation isolates failures on the compute and memory servers, enabling fast compute recovery. On the one hand, memory servers neither contain a CPU nor run any complicated software, hence they rarely fail and do not disrupt compute servers. On the other hand, a failed compute server need not stop the application entirely, as memory is available to the other compute servers. In addition to fast recovery, maintaining a recoverable state in DM is significantly faster with direct RDMA accesses or CXL (as opposed to RPCs), thereby maximizing performance.

Disaggregation proves crucial for key-value data stores (KVSes) that seek high availability. Typically, KVSes are partitioned, and these partitions are distributed between numerous servers within the datacenter, akin to a distributed shared memory system [59]. Crucially, the increasing rate of failures in modern datacenters poses a substantial challenge to the availability of these data partitions, necessitating fast recovery. Recovery, however, relies on fault-tolerant transaction protocols that replicate data across multiple servers for durability while ensuring the atomicity of data changes across replicated partitions (meaning that changes applied to one replica will be consistently visible in all other replicas [59, 30, 39]).

Traditionally, KVS data replicas are co-located within monolithic servers that handle both computation and data. However, this approach has a drawback: Memory failures can occur whenever a server crashes, thus reducing replica availability. One of the consequences of this is that, upon detecting a failure, recovery often necessitates stopping the data store entirely to update the new configurations of replicas (which is not conducive to achieving fast recovery). In contrast, disaggregated memory eliminates these limitations by isolating memory failures, allowing for fast recovery from compute failures, which are the predominant cause of failures in data stores [26]. Consequently, memory disaggregation can be leveraged to design performant key-value data stores that offer high availability (we refer to these data stores as *disaggregated key-value stores* or *DKVSes* here onward).

1.3 This Thesis

1.3.1 Problem Statement

We have thus far argued that emerging memory technologies surprisingly offer the opportunity to achieve fast recovery with maximum performance. However, these new memory devices come with different architectural attributes. Consequently, architecting datacenters with emerging memory introduces new semantic challenges. First, architects need precise semantics for end-to-end correctness, including recovery (Consistency Models). Then, architects need new low-level primitives to correctly and efficiently enforce these prescribed models. Failure to specify or enforce the precise consistency semantics not only complicates correctness but also hampers programmability and performance. In this section, we discuss the challenges associated with each emerging memory technology.

1.3.1.1 Non-Volatile Memory

Applications access non-volatile memory (NVM) through processor's load/store memory interface (as opposed to traditional I/O-based disks). Therefore, architecting datacenters with NVM requires new abstractions to reason about *persistent memory accesses*.

Regular memory accesses, due to complexities such as caching and concurrency in modern processors, require ordering that is governed by traditional shared-memory consistency models [79]. These models further bridge the gap between system architects and programmers by acting as a contract, wherein the architects guarantee specific correctness properties if programmers follow a set of rules defined by the model. However, traditional shared-memory consistency models, or the protocols that enforce them in server CPUs, fail to work with failures, posing a significant threat to correctness.

Ordering Challenges. NVM necessitates precise consistency models to control the order of memory updates to NVM (often referred to as Persistency Models). These models ensure end-to-end correctness of applications including recovery in the presence of failures. Introduced by Pelly et al [87, 175], persistency models remain an active area of research. Owing to recovery, these models introduce additional ordering, which further complicates programmability, or otherwise leads to significant performance losses [88].

Problem: Persistency For LFDs. Recall that coupled with NVM, log-free data structures (LFDs) can provide fast recovery. Alas, we observed that state-of-the-art per-

sistency models are neither sufficient nor optimal for recovering LFDs in NVM. These models lead to correctness bugs, making LFDs unrecoverable, and induce prohibitively large performance overhead.

1.3.1.2 Disaggregated Memory

Recall that disaggregated memory (DM) presents the opportunity for enabling fast recovery in key-value data stores (KVSes). Correct and fast recovery in these data stores rests on transaction protocols that ensure atomicity of a group of operations (like insert, delete, update, search), while hiding underlying complexity like replication from the application programmers. However, transaction protocols in this setting can only operate with one-sided RDMA messages, as RPC-style protocols do not work with passively disaggregated memory.

Ordering Challenges. One-sided accesses, in the absence of RPCs, expose critical low-level ordering of macro operations in transaction protocols to the high-level architectural interface. Therefore, system architects should carefully enforce transactional semantics with one-sided accesses, which otherwise pose a threat to correctness. Moreover, one-sided ordering is not only crucial for ensuring correctness but also plays a critical role in harnessing the performance potential of these new memory technologies. It allows architects to optimize performance based on the specific characteristics of one-sided accesses.

Problem: One-Sided Transactions. We have observed that there are no correct and recoverable one-sided transaction protocols. The only state-of-the-art one-sided protocol, FORD [219], which was developed in parallel to this thesis, has overlooked recovery, thereby leading to recovery bugs, while inducing prohibitively large recovery time.

(Additionally, previous disaggregation approaches, such as storage disaggregation, while addressing a similar availability problem, still rely on software Remote Procedure Calls (RPC) and have not been extensively studied with transaction or one-sided protocols due to various limitations [182]; These approaches are primarily tailored to distributed file systems or disk-based block storage, and therefore cannot be directly applied to the context of disaggregated memory or key-value stores [155, 25, 84].)

1.3.1.3 Summary

Emerging memory technologies are promising but share one challenge in common: Their unique architectural attributes, which drastically differ from traditional memory devices, expose critical low-level ordering to high-level application (or software) interfaces, presenting a semantic challenge. System architects typically understand end-to-end correctness through consistency models. Although abstract specifications of these models are well-understood, realizing them in practice with emerging memory is an incredibly hard challenge. This complexity affects not only correctness but also performance and programmability, presenting a significant challenge in harnessing the full potential of these technologies.

1.3.2 Research Objectives

The primary goal of this thesis is to investigate and address the challenges of achieving fast, correct, and performant recovery in modern datacenter architectures. We focus on leveraging two emerging memory technologies, nonvolatile memory (NVM) and disaggregated memory (DM). While these technologies offer opportunities for fast recovery and high performance, they also introduce new semantic challenges, which present significant barriers to the realization of their full potential. Our objective is to address these challenges that arise when architecting datacenters with emerging memory technologies. Our specific objectives are as follows.

Consistency Specification. We aim to define precise consistency models that formally specify end-to-end correctness guarantees for applications using emerging memory technologies within datacenters.

Low-Level Mechanisms. We seek to explore low-level primitives tailored to emerging memories that efficiently enforce the prescribed consistency models.

Testing and Validation. We aim to create robust testing frameworks to validate overall correctness, including recovery mechanisms.

Programmability. We aim to maintain programmer productivity (programmability) by minimizing exposure to low-level complexities in the consistency models, as subtle complexities in consistency models often introduce new ordering rules, hindering programmability.

These objectives collectively aim to unlock the potential of emerging memory for achieving fast, performant, and correct recovery in the datacenter architectures of the

next generation.

1.3.3 Scope

We further limit the discussion of this thesis to the following aspects.

First, we assume fail-stop failures (non-byzantine) [184]. A failed process or server is not allowed to continue with the failed components. This is the most common failure model in modern datacenters.

Second, we assume that there is one single datacenter rather than multiple datacenters. This approach allows us to concentrate on common failure scenarios, as fault tolerance complexity does not significantly increase with multiple datacenters. In fact, fault tolerance techniques developed for a single datacenter serve as foundational building blocks for similar techniques across datacenters.

1.4 Thesis Contributions

In this thesis, we explore the datacenter architecture with two emerging memory technologies, non-volatile memory (NVM) and disaggregated memory, to achieve our research objectives. In this section, we discuss our approach and its contributions. First, we start our discussion with non-volatile memory (NVM), and then we extend the discussion to disaggregated memory.

1.4.1 Lazy Release Persistency (LRP)

We address the challenges of architecting datacenters with NVM with the following approach: First, we rectify the critical ordering requirements for correct and recoverable lock-free data structures (LFDs). Second, we show that state-of-the-art persistency models are neither sufficient nor optimal for enforcing correct LFD semantics in the presence of failures. Third, strengthening the existing models, we introduce precise specifications of a novel persistency model –Release Persistency (RP) [55, 54]– that provides a consistency guarantee for LFDs on what remains in non-volatile memory upon failure. Next, to efficiently enforce RP, we propose a novel microarchitecture mechanism - Lazy Release Persistency (LRP) [55]. Finally, through extensive evaluation with standard LFD benchmarks, we demonstrate that LRP achieves fast recovery while incurring minimal overhead on performance.

Programmability. New consistency semantics (or persistency models) impact programmability as they introduce new ordering rules. One of the key advantages of Release Persistency is its compatibility with an already established programming model: Sequential Consistency for Data Race Freedom (DRF-SC). These models are already supported in popular programming languages like Java and C++. For instance, the programs written using standard C++ libraries can be directly compiled into their RP-based recoverable version without compromising programmability.

1.4.2 Pandora

We continue our contributions with disaggregated memory. Recall that disaggregation enables the possibility of rendering high availability, but achieving high availability hinges on correct and recoverable one-sided transaction protocols. First, we rectify one-sided ordering in disaggregated memory systems and their impact on transaction protocols. Second, we show that the state-of-the-art one-sided transaction protocol has overlooked one-sided ordering, leading to critical correctness and recovery bugs. Third, we present a highly available and recoverable transaction protocol– Pandora– which is specifically designed to achieve fast recovery in disaggregated key-value stores (DKVSes).

Pandora is the first one-sided transactional protocol that ensures correct, non-blocking, and fast recovery in DKVSes. Pandora's fast recovery is based on two key innovations: (i) Implicit Latch Logging (ILL) and (ii) end-to-end RDMA-based idempotent recovery algorithm. We implement Pandora using fully one-sided RDMA operations while avoiding other hardware complexities associated with disaggregation. Our implementation artifacts show that Pandora achieves fast recovery while incurring minimal interruption to the services.

Programmability. Pandora does not compromise programmability in the datacenter as we comply with the transactional semantics. Transactions are the de facto programming model in many state-of-the-art data store applications.

1.4.3 DART

One of the key challenges of designing one-sided transaction protocols is validating their end-to-end correctness, including recovery. For instance, how do we know that Pandora's recovery is correct? Existing techniques for testing generic applications

and transaction protocols are often limited to random testing and significantly impact programmability and scalability.

We propose a novel target litmus-testing framework–DART–for validating end-to-end correctness of transaction protocols with recovery. We use DART's target-testing abilities to find several critical bugs in Pandora, which we fix iteratively, highlighting the need for efficient testing methods for transaction protocols. Crucially, DART operates as a black-box testing framework, eliminating the need for programmers to invest additional effort or possess knowledge of the underlying complexities.

1.5 Summary

Emerging memory technologies, specifically non-volatile memory and disaggregated memory, offer a promising pathway to address increasing failures in modern datacenters with fast and performant recovery. However, this opportunity comes with new design and semantic challenges. When architecting datacenters with emerging memory, first, we need precise consistency models to reason about end-to-end correctness under failures, second, we need efficient architectural primitives (low-level mechanisms) to implement these models in the datacenter, and third, we need new methods for validating the end-to-end correctness, including recovery. In this thesis, we address the challenge while taking advantage of the opportunity for performance and availability.

This thesis is organized as follows. In Chapter 2, we establish the required background material. In Chapter 3, we present a new consistency model and an efficient microarchitecture implementation for NVM-based datacenter architecture. In Chapter 4, we describe a recoverable and highly available one-sided transaction protocol targeting disaggregated memory data stores. In Chapter 4, we introduce a new litmus-testing framework for validating the end-to-end correctness of transaction protocols. Our artifacts based on standard benchmarks demonstrate that the proposed solutions achieve fast and correct recovery while incurring minimal performance overhead. Finally, in Chapter 6, we summarize the dissertation and present some lessons that we learn during the course of this dissertation.

Chapter 2

Background

In this chapter, we lay the foundation by introducing the background materials essential to this thesis. First, we offer an overview of modern datacenter architecture, the failure model, and fault tolerance(§2.1). Second, we examine emerging memory and its influence on modern datacenter architecture (§2.2). Finally, we present a concise overview of relevant consistency models and how they respond to failures. (§2.3).

2.1 The Datacenter Architecture

In this section, we outline the key assumptions underlying our study, focusing on the architecture of datacenters and the failure model. A datacenter consists of interconnected servers, each equipped with CPU, memory, and storage resources (as depicted in Figure 2.1(a)). These servers run standard operating systems and software libraries to manage their resources and are interconnected through a network of switches and routers [27]. Datacenters are often spread across a large geographical area or even multiple global sites. Additionally, to enhance reliability and efficiency, datacenters deploy backup power supplies and cooling systems.

Network. Datacenter servers primarily communicate with each other using Remote Procedure Calls (RPCs). Each server is equipped with a Network Interface Card (NIC) that connects it to the datacenter network fabric. These servers employ standard transport layer protocols such as TCP/IP or RDMA (Remote Direct Memory Access) to send messages via the NICs. While protocols like TCP/IP enable send-receive (two-sided) message exchanges, RDMA-based technologies provide both two-sided and direct one-sided messages, enabling direct memory access to remote servers. In this thesis, we focus primarily on RDMA-based networking due to the significance of

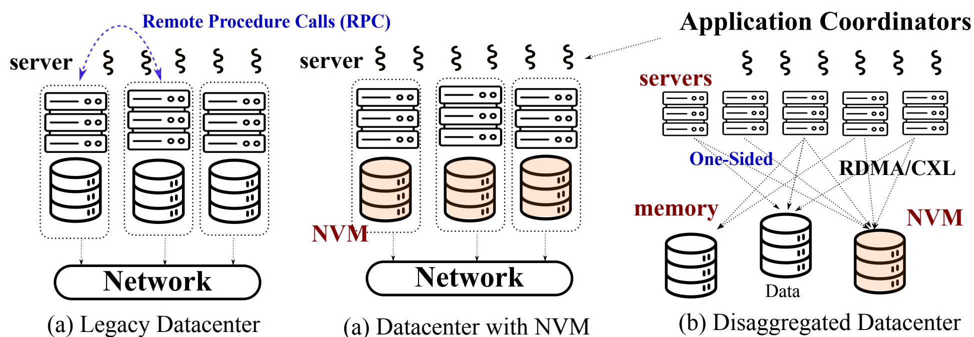


Figure 2.1: The Datacenter Architecture

RDMA's one-sided API, particularly in the context of exploring disaggregated memory, which is discussed in subsequent sections.

Failure Model. Failures can occur in any part of the datacenter. For this thesis, we have adopted a fail-stop failures model [184], where a process or server is assumed to have failed without exhibiting *Byzantine* behavior. Additionally, we consider a partially synchronous network model [63], where message transmission latency has an upper limit, albeit unknown in advance. This model strikes a balance between fully synchronous (where message delays and processing times are strictly bounded) and fully asynchronous (where there are no timing assumptions) network models. It further acknowledges the practical constraints of real-world networks, where messages may experience variable delays due to factors like network delays, system load, and other unpredictable factors.

2.1.1 Recovery

Failures disrupt datacenter servers, tapering their ability to respond to service requests (Availability). Recovery is the mechanism through which architects maximize availability in the datacenter. For instance, high availability consequently ensures responsiveness even amid server failures in the datacenter. Recovery is challenging, as it requires additional measures.

Durability. First, it's essential to store application data in a fault-tolerant manner to prevent permanent loss (Durability) '[30, 100]. Architects typically deploy two techniques for durable data storage in the datacenter: *persistence* and *replication* [30, 193, 65, 96, 97, 76]. With persistence, data is retained in a persistent storage medium like a Hard-Disk, while with replication, data is duplicated across multiple servers. In the case of persistence, the application can restart the failed server while

temporarily blocking other servers. Conversely, replication allows the failed portion of the application to migrate to an active server and continue serving client requests [40, 135, 77, 202, 198].

There are also various replication approaches, including two-phase commit [30, 31, 94], primary-backup replication [39, 14, 202, 198], and quorum-based replication [133, 140, 170, 21, 116]. The selection of the best approach depends on the real world setting and the application requirements.

Failure Atomicity. Second, recovery mandates failure atomcity [30, 69, 194], because, just like data, computational logic also fail in the event of server failures, resulting in partial effects on the data. Recovery ensures that every client request is processed with exactly-once semantics, in all-or-nothing style, avoiding any partial effects on data (*Failure Atomicity*) [30, 29].

For example, imagine a seemingly trivial example of a banking application running on a data store. Lets assume Mary want to transfer 1000$ to her friend Bob. When the data store receives this request, the application logic should first check if Mary has enough balance to carry out the transaction, then reduce the amount from Mary's bank account, and finally add the amount to Bob's bank account. If the server fails after the amount is deduced from Mary's bank account, the money will be lost forever. Crucially, there should be a mechanism to rollback the effects of operations executed before the failures (Failure Atomicity). Such a semantic is achieved by recovery algorithms that reinstate the data to a consistent state in execution.

Achieving such robust recovery is complex and has traditionally been tackled through two fundamental methods: *re-execution* and *logging*. This section provides a concise overview of these techniques, laying the foundation for a more detailed exploration in the following main chapters of this thesis.

Firstly, applications may choose to rerun client requests multiple times, even when faced with failures. However, not all applications can rely on this recovery approach. Specifically, only a specialized class of applications that adhere to idempotence can truly take advantage of re-execution [179, 114, 41]. It is important to note that this technique might introduce challenges to concurrency and programmability, which is why we refrain from adopting it.

Secondly, an alternative approach to failure atomicity is logging, where a set of operations is treated as executed in an all-or-nothing manner [100, 93, 160, 159]. Logging is widely accepted within the datacenter context. Applications use mechanisms such as *write-ahead-logging* (WAL) or checkpointing to restore or progress the application to

its last consistent state. There are two primary WAL schemes: undo logging, designed to roll back partial updates from interrupted executions, and redo logging, which rolls forward when an execution is disrupted before its completion. This atomicity-based recovery plays a pivotal role in important programming models such as transactions in the datacenter.

End-to-End Correctness. In addition to durability and failure atomcity, application must ensure the isolation between concurrent requests in the presence of failures. For instance, recovering from a failures should not lead to incorrect execution of other requests. These correctness properties are typically defined as a part of the consistency models in the datacenter. We will discuss consistency models in the Section 2.3 .

2.2 Emerging Memory Systems

In this thesis, we use two types of emerging memory technologies: non-volatile memory (NVM) and disaggregated memory (DM). In this section, we provide an overview of their fault-tolerant attributes and impact on datacenter architecture. First, we discuss nonvolatile memory, outlining its impact on persistence-based recovery (§2.2.1), and second, we discuss memory disaggregation and its impact on high availability in data stores (§2.2.2).

2.2.1 Non-Volatile Memory (NVM)

In this section, we outline our assumptions about non-volatile memory (NVM) technology we use in this thesis.

Various classes of NVMs are available in the market: Phase-change memory (PCM), resistive random-access memory (ReRAM), or magnetoresistive RAM (MRAM), and ferroelectric RAM (FRAM). In this thesis, we focus on PCM-based NVM, specifically Intel's Optane DC Persistent Memory Module (or just "Optane DC PMM") which can provide up to 512GB memory capacity. Crucially, these new memory devices retain data even when power is interrupted (persistent), therefore, they can be used as a fast alternative to disk-based storage, or as an additional memory tier between volatile memory and disk-based storage [212, 112] .

Typical disk-based storage in the datacenter, such as hard disk drives (HDD) or solid state disk drives (SDD), provides terabytes of capacity, but can only be accessible through the CPU's I/O (Input/Output) interface which has significantly high access

latency, typically ranging from a hundreds of microseconds to tens of milliseconds. Conversely, NVM provides fast persistent memory accessible within a few hundred nanoseconds, which is on par with regular volatile memory access latency [212, 112].

For instance, PMEM latency of sequential reads (random reads) and regular writes is 179 ns (304 ns) and 94 ns respectively, while its 81 ns and 86 ns for DDR5-based DRAM [112]. Additionally, Optane DC PMM provides caching within the NVM, further minimizing these latencies. This shows that NVM, though still at an early stage, can be purposefully leveraged as an alternative to slow storage devices. As we will discuss later in Chapter 3, this fast persistence can be used to enable efficient fault tolerance techniques that rely on persistence for recovery.

2.2.2 Disaggregated Memory (DM)

The other memory technology used in this thesis is disaggregated memory (DM). Memory disaggregation decouples memory from traditional monolithic servers, enabling memory sharing across server boundaries [191, 15, 145, 47, 164, 205]. State-of-the-art hardware technologies, RDMA or CXL [90, 139], provide network support for connecting passive memory to the servers' CPU cores through either I/O subsystems or direct load/store interface respectively. After careful consideration, we have chosen an RDMA-based passive disaggregated memory system for this thesis. This choice is based on RDMA's sub-microsecond latency with 100Gbps technology, offering flexibility in programming (this latency is anticipated to decrease further with the adoption of 400G networks and faster PCIe devices in the near future). On the contrary, CXL-based systems are still in the early development stage, and architects are unaware of potential performance challenges and programming complexities [9, 139]. At the time of this thesis, there is only one CXL-based product readily available on the market [183].

In this thesis, we assume an RDMA-based disaggregated memory system. In our setting, each server is dedicated to either compute or memory, referred to as compute node or memory node, respectively. Each compute node consists of a CPU and a small amount of local memory (a few MBs) for caching purposes and other OS functionalities, while the program memory is stored entirely in the memory nodes, typically holding terabytes of data. All compute nodes can access and share each memory node through RDMA connections established over a 100Gbps Ethernet network (i.e., RoCE).

A typical memory node neither has a CPU nor runs any software like OS; Therefore, memory is accessed only through the Network Interface Card (NIC) or special hardware.

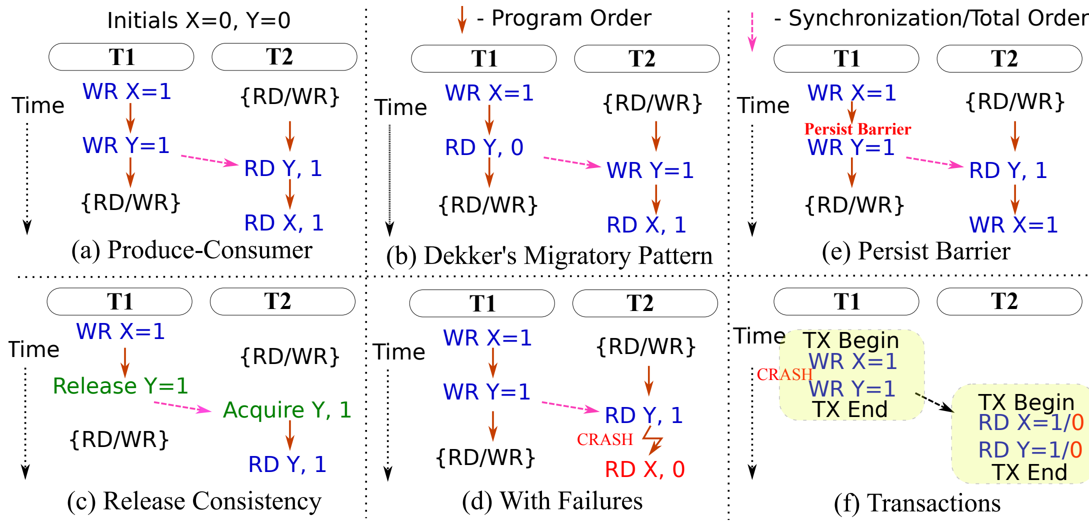


Figure 2.2: Consistency Models

However, since there are no actual memory nodes available in the market, in this thesis, we use regular servers for memory nodes while keeping their CPU idle, which does not violate our assumptions. In addition, RDMA offers a rich one-sided API that includes Read, Write, and Read-Modify-Write (RMW) commands, which is order of magnitude faster than traditional two-sided Remote Procedure Calls (RPCs).

Figure 2.1(c) illustrates the architecture we have discussed so far. Later, in Chapter 4, we leverage this RDMA-based disaggregated memory to achieve efficient fault tolerance in the datacenter.

2.3 Consistency Models

A consistency model plays a crucial role in abstracting the end-to-end correctness of the underlying architecture to application programmers, simplifying the complexity of low-level primitives. This implementation-agnostic nature allows programmers to reason about the correctness of their programs without delving into low-level implementation aspects like concurrency, replication, and recovery. Historically, consistency models have been pivotal in datacenters, acting as a contract between architects and programmers [5, 79].

In this section, we discuss several consistency models that are relevant to the discussion of this thesis. We start the discussion with traditional (shared-memory) consistency models: linearizability, sequential consistency, and release consistency [5, 79, 80]. These consistency models are the essence of high concurrency present in

today's CPUs.

2.3.1 Linearizability

Linearizability [108] is the most intuitive consistency model for programmers, ensuring that concurrent operations appear as if they occurred instantaneously at a single point in time (linearization point) in relation to other operations. This means that the effect of each operation becomes atomically visible in real-time somewhere between its invocation and response. Typically, coherence protocols deployed in modern CPUs are variants of linearizability. However, for shared-memory programs, linearizability represents a stricter consistency model.

2.3.2 Sequential Consistency

Sequential consistency (SC) [132] guarantees that the execution of operations on shared data appears in some sequential order (total order) that respects the program order for each concurrent operator. As opposed to linearizability, SC does not enforce the real-time order, striking a good balance between performance and programmability in shared-memory systems. A successor of SC dubbed Total Store Order (TSO) found in Intel's X86 CPUs further maximum performance by relaxing local happens-before order, specifically reads-after-writes order. Such models, however, complicates programmability and will be absorbed by more weaker models we discuss later.

Consistency semantics are typically discerned through well-constructed litmus tests. Consider the litmus tests depicted in Figure 5.1(a) and (b). These tests exemplify Sequential Consistency (SC) semantics with two prevalent shared-memory programming patterns: producer-consumer (Figure 5.1(a)) and migratory patterns with Dekker's algorithm (Figure 5.1(b)).

In these litmus tests, SC rigorously enforces both program order and total order. For instance, in the producer-consumer illustration, one thread (T1) first sets X to one, followed by setting Y to 1 to indicate the alteration in X. Until Y is assigned the value of 1, the concurrent thread T2 cannot read the updated value of X, which is 1. This mechanism aligns with the typical behavior of producer-consumer patterns, which necessitate enforcing read-to-read and write-to-write order to capture the underlying causality, which SC effectively accomplishes. Similarly, migratory litmus tests necessitate write-to-reads order to ensure critical sections, a requirement prevalent in synchronization algorithms like Dekker's, which SC strictly enforces.

One advantage of SC is that programmers are oblivious to the underlying ordering. However, SC is somewhat strict in enforcing all local happens-before order (program order), which might not be as performance-friendly. However, well-written (i.e., race-free) shared memory programs using proper synchronization do not require such tight ordering [6, 82]. The golden rule is that operations should be consistent by synchronization boundaries. This understanding has led to the development of more performant models, such as release consistency, which we will describe next.

2.3.3 Release Consistency

Release Consistency (RC) [82] is a fundamental shared memory consistency model that guarantees that if the programmer annotates all synchronization accesses, it mimics SC semantics [6, 82, 78]. For architects, RC is essential, as it allows reordering regular accesses, relaxing the local happens-before order (as opposed to SC). Additionally, RC provides one-sided synchronization accesses that can be distinguished from regular accesses, allowing for a more relaxed happens-before order between regular access and synchronization accesses.

Languages like C++ enable programmers to expose RC through language-level consistency models, such as the "Sequential Consistency for Data Race Free" (DRF-SC) [153, 36], which captures the semantics of RC-like synchronization accesses. Additionally, CPU vendors like ARM provide RC semantics in the ISA. For instance, ARMv8 has two one-sided barrier instruction, release and acquire, that can be used to annotate synchronization load/store accesses.

Consider the example in Figure 5.1(c), which provides the RC-compatible (annotated) version of the producer-consumer litmus test discussed earlier. In this scenario, the release store operation in T1 enforces the order with previous loads and stores, but not with subsequent instructions. Similarly, the acquire operation in T2 enforces the order with following loads and stores, but not with preceding instructions. It is important to note that operations before the release, as well as after acquire, can potentially be reordered, although that is not depicted in the example.

Formal Model. In the following, we provide a simplistic RC memory model for this thesis.

We use the following notation for memory events:

- $\mathbf{M_x^i}$: a memory operation (of any type) to address x from (hardware) thread i. The operation can be further specified as a read: R_x^i, a write W_x^i or with an identifier (e.g.

$M1_x^i$)

- $\mathbf{Rel_x^i}$: a release (release write or release-RMW) to address x from thread i.
- $\mathbf{Acq_x^i}$: an acquire (acquire read or acquire-RMW) to address x from thread i.

We use the following notation for ordering memory events:

- $\mathbf{M_x^i} \xrightarrow{\text{po}} \mathbf{M_y^i}$: M_x^i precedes M_y^i in program order.
- $\mathbf{M_x^i} \xrightarrow{\text{hb}} \mathbf{M_y^j}$: M_x^i precedes M_y^j in the global history of memory events, which we refer to as happens-before order ($\xrightarrow{\text{hb}}$).
- $\mathbf{Rel_x^i} \xrightarrow{\text{sw}} \mathbf{Acq_x^j}$: Acq_x^j *synchronizes with* the Rel_x^i, i.e., Acq_x^j reads the value from Rel_x^i and $i \neq j$.

We formalize Release Consistency using the following rules:

$\succ$ **Release one-way barrier semantics.** A memory access that precedes a release in program order appears before the release in happens-before: $M_x^i \xrightarrow{\text{po}} Rel_y^i \Rightarrow M_x^i \xrightarrow{\text{hb}} Rel_y^i$.

$\succ$ **Acquire one-way barrier semantics.** A memory access that follows an acquire in program order appears after the acquire in happens-before: $Acq_y^i \xrightarrow{\text{po}} M_x^i \Rightarrow Acq_y^i \xrightarrow{\text{hb}} M_x^i$.

$\succ$ **Program order address dependency.** Two memory accesses to the same address ordered in program order preserve their ordering in happens-before: $M1_x^i \xrightarrow{\text{po}} M2_x^i \Rightarrow M1_x^i \xrightarrow{\text{hb}} M2_x^i$.

$\succ$ **Release synchronizes with acquire.** A release that synchronizes with an acquire appears before the acquire in happens-before: $Rel_y^i \xrightarrow{\text{sw}} Acq_y^j \Rightarrow Rel_y^i \xrightarrow{\text{hb}} Acq_y^j$.

$\succ$ **RMW-atomicity axiom.** An RMW appears atomically (consecutively) in happens-before: $R_x^i \xrightarrow{RMW} W_x^i \Rightarrow R_x^i \xrightarrow{hb} W_x^i$ and there can be no memory operation from any thread M_y^j such that $R_x^i \xrightarrow{\text{hb}} M_y^j \xrightarrow{\text{hb}} W_x^i$.

$\succ$ **Read value axiom.** A read to an address always reads the latest write to that address before the read in happens-before: if $W_x^j \xrightarrow{\text{hb}} R_x^i$ (and there is no other intervening write W_x^k such that $W_x^j \xrightarrow{\text{hb}} W_x^k \xrightarrow{\text{hb}} R_x^i$), the read R_x^i returns the value written by the write W_x^j.

2.4 Memory Persistency Models

Traditional shared-memory consistency models that we have discussed are primarily designed to handle concurrent accesses in the presence of caching but are not explicitly designed to handle failures. Therefore, these models are not sufficient to ensure end-to-end correctness of new technologies like NVM.

As an example, consider the litmus test in Figure 5.1(e), depicting the producer-consumer pattern under SC semantics but with potential failures. Suppose that T2 reads Y's updated value as part of the consumer, but the server crashes before it can complete the subsequent operation. After restarting, T2 resumes execution. In this situation, T2 could pathologically read the initial value of X (or zero) because X's value, which is kept in either cache or memory, is lost, revealing that SC is insufficient to handle such failures.

Persistence models [175, 87] are a new class of consistency models that are specifically designed to address the complexities introduced by persistent memory and guarantee the desired correctness properties, including recovery even in the face of failures. In this section, we briefly look at the concept of persistency models.

Memory persistency is still an active research area. Persistence models guarantee the correctness of application programs with persistent memory accesses [88]. In particular, these models govern the memory order that ensures correct execution and recovery in the presence of failures. However, these new models have severe implications on performance and programmability [37, 165].

Strict persistency models either bypass CPU caches using non-temporal instruction or use a new instruction to durably send every store to NVM before being released into CPU caches, which can significantly deteriorate concurrency and performance [87, 175]. On the other hand, relaxing the persist order, though performant, can severely complicate correctness and programmability [175]. The complexity of the latter approach is significantly higher, so architects have opted to go with strict persistency models [87].

Architects must carefully examine new persistence models with minimal ordering to unleash fast persistence in NVM, a crucial factor in designing efficient fault tolerance in datacenters. We will discuss more about the challenges in memory persistency in Chapter 3.

2.5 Transactional Consistency Models

Transactions combine multiple standard operations, such as insertions, deletions, updates, and search (reads), into atomic units. SS is subsequently designed to maintain the atomic view of data under any circumstance, ensuring end-to-end correctness, including recovery of transactions in the event of failures.

Consider the final example shown in Figure 5.1(f). In this example, we use the same producer-consumer pattern that we used with sequential consistency (SC) but

as a transaction. Application programmers can annotate a group of operations as a transaction with *tx_begin* and *tx_end*, and the atomicity of the transaction is preserved in the presence of failures. Now assume that the server crashes during execution; as shown in the figure, SS guarantees that T2 sees either the old values of X and Y (all zeros) or new values (all ones), but nothing in between.

2.5.1 Strict Serializability

Strict serializability (SS) [174, 187] is a rigorously formalized consistency model widely used in production data stores. It operates as a transactional consistency model, offering strong ACID (Atomicity, Consistency, Isolation and Durability) properties in the presence of failures.

In formal terms, *serializability* ensure that all transactions appear to occur in total order (i.e., serial) as if they were executed one at a time in isolation. Serializability, however, does not guarantee real-time ordering. In contrast, a stronger version of serilizability – Strict Serializability(SS) – ensures that the effects of transactions appear atomically in real time. In this section, we provide a formalization of *strict serializability* with two important properties: total ordering (i.e., serializability) and real-time ordering.

2.5.1.1 Serializability

Serializability [31, 174, 7] guarantees that the outcome of concurrent transactions is equivalent to a serial execution (total order) of those transactions, ensuring that the final state of the database remains consistent and reflects a valid sequential order of execution. In this thesis, we use Adya's history-based formalism of serializability [7, 53].

Definition 2.5.1. A history H over a set of transactions consists of two parts: (i) a partial order of events E that reflects the operations (e.g., read, write, abort, commit) of those transactions; and (ii) a version order, $<<$, that totally orders committed object versions.

Definition 2.5.2. We consider three kinds of direct read/write conflicts:

- **Directly write-depends** T_i writes a version of x and T_j writes the next version of x ($T_i \xrightarrow{ww} T_j$)
- **Directly read-depends** T_i writes a version of x that T_j then reads ($T_i \xrightarrow{wr} T_j$)
- **Directly anti-depends** T_i reads a version of x, and T_j writes the next version of x ($T_i \xrightarrow{rw} T_j$)

Definition 2.5.3. We say that T_j start-depends on T_i (denoted as $T_i \xrightarrow{sd} T_j$), if $c_i <_t b_j$, where c_i denotes T_i's commit timestamp and b_j T_j's start timestamp, i.e., if T_j starts after T_i commits.

Definition 2.5.4 (Direct Serialization Graph)**.** Each node in the direct serialization graph $DSG(H)$ arising from a history H corresponds to a committed transaction in H. Directed edges in DSG(H) correspond to different types of direct conflicts. There is a read/write/anti-dependency edge from transaction T_i to transaction T_j if T_j directly read/write/antidepends on T_i.

Definition 2.5.5. The Started-ordered Serialization Graph SSG(H) contains the same nodes and edges as DSG(H) along with start-dependency edges.

Definition 2.5.6. Adya's thesis identifies the following phenomena:

- **G0:** Write Cycles DSG(H) contains a directed cycle consisting entirely of write-dependency edges.
- **G1a:** Dirty Reads H contains an aborted transaction T_i and a committed transaction T_j such that T_j has read the same object (maybe via a predicate) modified by T_i.
- **G1b:** Intermediate Reads H contains a committed transaction T_j that has read a version of object x written by transaction T_i that was not T_i's final modification of x.
- **G1c:** Circular Information Flow DSG(H) contains a directed cycle consisting entirely of dependency edges.
- **G1:** $G1a \vee G1b \vee G1c$.
- **G2:** Anti-dependency Cycles DSG(H) contains a directed cycle having one or more anti-dependency edges.

Definition 2.5.7. Adya defines serializability in terms of the phenomena in Definition 2.5.6:

$Serializability \equiv \neg G1 \wedge \neg G2$

2.5.1.2 Real-time Ordering

Strict serilizability enforces a strong serilizable order that respects real-time ordering. Transactions appear atomically in real time. In the similar spirit to DSG, we can define a new graph with real-time.

Definition 2.5.8 (Real-order edge)**.** If a transaction T_i commits before T_j executes its first event in *real-time*, we add a real-order edge from T_i to T_j ($T_i \xrightarrow{rt} T_j$)

Definition 2.5.9 (Realtime Serialization Graph)**.** Real-time Serialization Graph or RSG(H) has the same nodes and edges as the DSG(H) along with real-order edges to capture the order of non-overlapping transactions. (These edges force non-overlapping transactions to be serialized in the order in which they executed in real-time.)

Definition 2.5.10. Strict serializability is defined as the level that proscribes G1 and G2 when phenomena in Definition 2.5.6 are defined on the RSG instead of the DSG.

Chapter 3

Lazy Release Persistency

3.1 Introduction

The advent of fast non-volatile memory (NVM) has enabled the possibility of recovering from a system crash while incurring minimal overhead during program's normal operation [1, 44, 50, 51, 113, 166, 204, 186]. Program recovery, however, hinges on primitives that control the order in which data becomes persistent. What primitive(s) offer a programmable interface while allowing for an efficient implementation at the hardware level?

The question is the subject of an ongoing debate. Should languages support *failure-atomicity* for a group of writes, or should languages forego atomicity and support *only ordering* between individual word-granular writes?

Kolli et al. [128] make a case for *only ordering*, arguing that it is more general and performance-friendly when compared to failure-atomicity which requires logging. They reason that because future processors are likely only going to guarantee atomicity of individual persists, a library that provides failure-atomicity can be used when necessary. They then propose *acquire-release persistency* (ARP), a language-level persistency model that extends C++11 by treating its release/acquire annotations as one-sided persist barriers. They also propose a hardware mechanism for enforcing these one-sided barriers efficiently. Thus, the key to ARP's performance is its one-sided barriers that attempt to precisely enforce the orderings intended by the programmer.

In subsequent work, however, Gogte et al. [86] make a case for failure-atomicity, arguing that the absence of failure-atomicity in ARP (and indeed any ordering primitive) makes reasoning about recovery extremely cumbersome.

Not all programs require failure-atomicity, however. In fact, an important class of

nonblocking data structures [34, 56, 72, 111, 169, 185] is designed specifically to avoid atomic regions. Recovery from a crash comes for free, aka *null recovery*, as long as writes persist in the order in which they become visible [111, 203, 35]. The emergence of NVM has sparked interest in these *log-free data structures* (LFDs) [56] primarily because such programs enable recovery without the overhead of logging.

Therefore, while we concur with Gotge et al. [86] that failure-atomicity simplifies recovery in the general case, we argue that languages must *also* offer efficient ordering primitives for supporting LFDs.

However, we identify that ARP's one-sided barriers are not strong enough to enable recovery in LFDs. Consider Figure 3.1 that depicts an execution history of a concurrent log-free linked list. Thread T0 first prepares node A1 for insertion by writing to its fields (Figure 3.1a). Then, it links A1 with the rest of the list via a single atomic Compare-and-Swap (CAS) instruction (Figure 3.1b). Note that from a consistency standpoint, the CAS must have release semantics to ensure that the writes to A1 become visible before the link is updated. To enable recovery, persistency must mirror visibility: the writes to A must persist before the CAS persists (Figure 3.1d). However, ARP's one-way barrier does not provide this guarantee. (As shown in Figure 3.1e, it only ensures that writes to A1 persist before writes to B2 from the acquiring thread persists). Therefore, to enable recovery, the programmer must place full persist barriers before the release and after the acquire (Figure 3.1f). Alas, the full barrier requirement annuls ARP's performance benefits which stem from its one-sided barriers.

In this thesis, we propose strengthening the one-sided barrier semantics of ARP to enable recovery of LFDs. The resulting persistency model, dubbed *Release Persistency* (RP), ensures that any two writes that are ordered by the consistency model also persist in that order. Thus, the persistency model guarantees that the NVM will hold a consistent-cut of the execution upon a crash, thereby satisfying the criterion for correct recovery of an LFD [111].

We then propose an efficient microarchitectural mechanism for enforcing the one-sided barrier semantics of RP. Going back to Figure 3.1d, the challenge is to enforce $W_1 \xrightarrow{p} Rel \xrightarrow{p} W_4$, without enforcing either $W_1 \xrightarrow{p} W2$ or $W_3 \xrightarrow{p} W_4$. (The relation $\xrightarrow{p}$ denotes the persist order). We observe that efficiency necessitates a buffered implementation in which persistency is decoupled from visibility [51, 117, 129]. Taking inspiration from lazy release consistency [124], a protocol from the DSM literature that enforces RC lazily, we propose lazy release persistency (LRP) for enforcing RP's one-sided barrier semantics lazily. In a nutshell, on an acquire LRP detects the match-

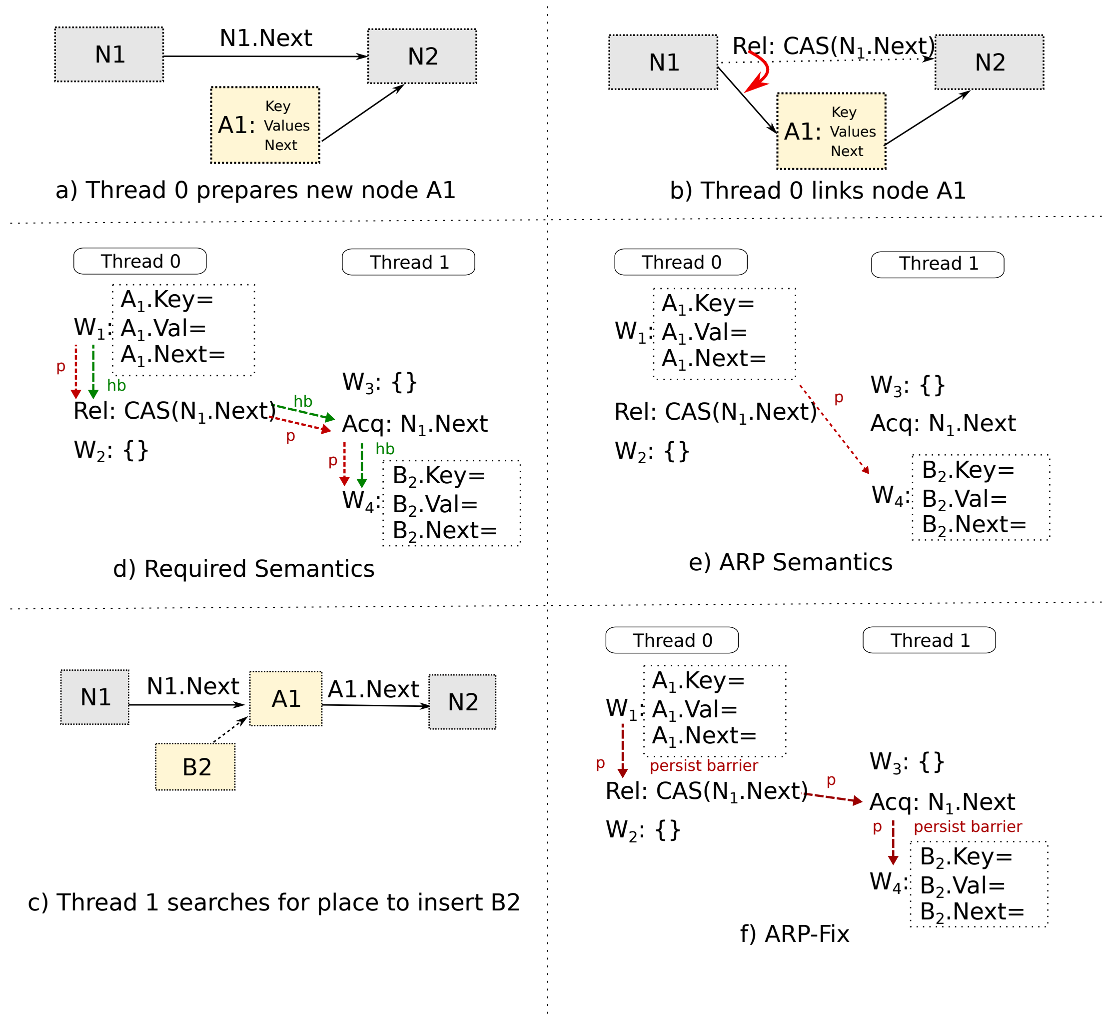


Figure 3.1: (a,b) Thread 0 inserts A1 in a log-free linked list. (c) Then, Thread 1 attempts to insert B2. (d) shows the required persistency semantics for an insert. (e) shows the semantics provided by ARP and (f) shows how ARP can match the required semantics.

ing release via the coherence protocol and enforces the release-side persist ordering ($W_1 \xrightarrow{p} Rel$) lazily before performing the acquire.

3.1.1 Contributions

- We have argued thus far that languages must offer efficient ordering primitives, in addition to failure-atomicity, for supporting the important use case of LFDs.
- We observe that ARP's one-sided barriers—their semantics as well as implementation—are not strong enough to enable recovery in LFDs, which necessitates the inclusion of the relatively inefficient full barriers (§3.3)

- We propose strengthening ARP's one-sided barrier semantics. We argue that the resulting model RP enables correct recovery of LFDs upon a crash (§3.4).
- We propose LRP, a microarchitectural mechanism for efficiently enforcing RP's one-sided barriers (§3.5) .
- Our experiments on 5 commonly used LFDs suggests that LRP provides a 16%-55% (average 43%) performance improvement over the state-of-the-art full barrier, while enforcing RP (§4.5).

3.2 Preliminaries

In this section, we discuss persistency models with a focus on variants of (buffered) epoch persistency (§3.2.2). We then discuss log-free data structures (LFDs) and describe the actions required for their crash-recovery (§3.2.3). But before diving into persistency, we first discuss consistency since the two are closely intertwined.

Without loss of generality, we assume a simple variant of Release Consistency with a total order on all memory events, similar to what is supported by the ARMv8 and RISC-V ISAs [18, 208]. To focus on our ideas and not get bogged down by memory model intricacies, we assume the language-level model is identical to the ISA-level model.

3.2.1 Release Consistency

A consistency model specifies how memory operations are globally ordered. This global memory order specifies what value a read must return: a read returns the value of the most recent write before it in the global memory order. Release Consistency (RC) [81] allows for writes to be tagged as releases and reads as acquires, which have implicit one-side barrier semantics. Specifically, memory operations before a release appear in the global order before the release, and memory operations after an acquire appear in the order after the acquire. Furthermore, most consistency models support read-modify-writes (RMWs) which are essential for achieving synchronization [23, 107].

For the rest of this work, we will use the simplistic RC memory model that we discussed in Section 2.3.

3.2.2 Persistency Models

In a manner analogous to consistency models, Pelly et al. [176] introduce the notion of memory persistency, which specifies a global order in which writes can persist (i.e., *persist order*).

Persist notation. We use the following notation to denote that write W_x^i appears after write W_y^j in persist order: $W_x^i \xrightarrow{\text{p}} W_y^j$. To put it succinctly, W_y^j can persist only after W_x^i has persisted.

Buffered Epoch Persistency (BEP). BEP allows the programmer to place *persist barriers*, demarcating the program in *epochs* [51]. BEP then uses the epochs to enforce that for any two writes W_x^i, W_y^i, if $W_x^i \xrightarrow{\text{po}} W_y^i$ and the writes belong to different epoch e_k, e_l, then $W_x^i \xrightarrow{\text{p}} W_y^i$. BEP also involves an inter-thread component. When there is an inter-thread shared memory dependency between two threads the writes from the source epoch would have to be persisted before the writes from the target epoch. We note that BEP is a performance-oriented variant of the stricter epoch persistency (EP). BEP improves upon EP by decoupling persistency and visibility through buffering of writes.

3.2.2.1 Persist Barrier Implementations

BEP can be enforced via persist barriers. We classify prior work on persist barriers into two classes based on how writes are buffered: 1) *cache-based* implementations that use the hardware caches to buffer writes and 2) *persist-buffer-based* implementations, that enqueue all writes in a global FIFO, called *persist buffer*. Below we describe the two classes.

Cache-based. Cache-based implementations [51, 117] buffer writes in the hardware caches and enforces the epoch orderings by keeping track of each cache-line's epoch. Persisting a cache-line with epoch e_k, triggers the persist of all currently buffered writes from older epochs. A persist is triggered from a *conflict*; there are two types of conflicts.

$\succ$ **Intra-thread conflicts.** There are two sources of intra-thread conflicts: 1) evicting a cache-line due to demand access and 2) attempting a write with epoch e_k on a cache-line with an older epoch-id.

$\succ$ **Inter-thread conflicts.** These are caused by inter-thread shared memory dependencies. Such dependencies mandate the enforcement of persist ordering between epochs

of two different threads. Inter-thread conflicts can be resolved by blocking the target thread until the source epoch persists [51]. It is possible to enforce these in a lazy manner [117] although that comes with the complexity cost of avoiding deadlocks if there are cyclic dependencies.

Note that while conflicts often trigger multiple persists on the critical path of execution, state-of-the-art cache-based implementations mitigate this overhead, through *proactive flushing* [117], a technique that starts flushing an epoch as soon as its execution completes.

Persist-buffer-based. Alternative persist barrier implementations [129, 163] buffer writes in a persist-buffer: a FIFO queue in the cache hierarchy, that orders writes. Writes enter the persist buffer in program order and before becoming visible to other threads, i.e., a write cannot be read by a remote thread unless it has entered the persist buffer. Notably, these implementations either enforce a global order across independent epochs belonging to different threads [129] or statically partition the persist-buffer entries across threads [163]. As a result, the persist buffers are not optimally utilized, and a large number of persist operations happen in the critical path of other read/write requests.

LRP approach. Persist buffers simplify the design, avoiding the complexity of tracking conflicts inside the caches. However, this approach limits the benefits of buffering as the buffers (which are typically smaller than caches) are not optimally utilized. For this reason, we choose to implement RP using the cache-based approach (§ 3.4) because, despite its complexity, it is likely more efficient.

3.2.3 Log-free data structures (LFDs)

To ensure correctness, operations that modify a data structure must be atomic. Often atomicity is achieved through atomic code regions protected by locks. However, an important class of *nonblocking* data structures are designed to explicitly avoid locks. To achieve this, such data structures carefully bake their atomicity into a single instruction (typically a Compare And Swap, i.e., CAS). Collapsing the atomic region into a single instruction eliminates the need for locks. For instance, in the example of Figure 3.1, Thread T0 first creates a node privately and then it atomically links the node with the linked list through a single CAS instruction.

The intention of this design pattern is to avoid blocking, i.e., avoid states in which a thread is unable to make progress without the cooperation of one or more peers [185].

In doing so, nonblocking data structures also eliminate the need for failure atomicity of a group of writes, since the atomicity is now incorporated into a single instruction, there is no longer need to persist multiple instructions atomically. Therefore, nonblocking data structures are *log-free*, as they do not require the logging mechanisms that are typically associated with failure atomicity.

To recover a program's progress, the PM must be kept in a consistent state. For an LFD where failure atomicity is not required, a consistent state can be achieved by simply ensuring that the PM writes a *consistent cut* of the program's execution, i.e., the persistency order need only mirror the happens-before relations mandated by consistency [168]. Indeed, Izraelevitz and Scott [111] prove that log-free programs can be recovered without any effort (i.e., null recovery), if what remains on the PM after a crash is a consistent cut of the program's execution. In this thesis, we aspire to provide null recovery for log-free programs, through an RC-based persistency model (RP) and an efficient mechanism (LRP) for enforcing RP.

3.3 Limitations of ARP

Gotge et al. [86] argue that a persistency model based on only ordering (such as ARP) is unsatisfactory because the lack of atomicity guarantees makes recovery cumbersome. While agreeing with Gotge et al. for general programs, we argue for their utility for LFDs, which can be recovered without any effort after a crash (i.e., null recovery), as long as NVM reflects a consistent cut of the program's execution.

Design goal. In order to maximize performance, while allowing for null recovery, we set the following design goal: the persistency model must mirror the RC semantics, without exceeding them. The key requirement to match RC semantics is treating releases and acquires as *one-way persist barriers*, in the same manner as RC treats them as one-way barriers.

Alas, ARP [128], the only RC-based persistency model with one-sided barriers falls short of that goal. In the rest of this section, we first describe ARP's semantics (§3.3.1) and implementation (§3.3.2) focusing on how ARP fails to achieve our design goal, and then we motivate the need for a new persistency model that can rectify ARP's shortcomings (§3.3.3).

3.3.1 ARP semantics

ARP [128] is a language-level persistency model with explicit release and acquire annotations. These semantics comprise the *ARP-rule*, which we define below.

ARP-rule. When a release synchronizes with an acquire, all writes that precede the release must persist before writes that follow the acquire:

$$W_y^i \xrightarrow{\text{po}} Rel_x^i \xrightarrow{\text{sw}} Acq_x^j \xrightarrow{\text{po}} W_z^j \Rightarrow W_y^i \xrightarrow{\text{p}} W_z^j$$

3.3.1.1 ARP semantics shortcomings

We note that the ARP-rule does not mirror the happens-before relations of RC and thus it is unable to provide null recovery as it does not preserve a consistent cut in the NVM. For instance, RC mandates that if a release is visible, all preceding writes must be visible, too, i.e., $W_y^i \xrightarrow{\text{po}} Rel_x^i \Rightarrow W_y^i \xrightarrow{\text{hb}} Rel_x^i$. However, ARP allows for a release to persist before all preceding writes have persisted; i.e., $W_y^i \xrightarrow{\text{po}} Rel_x^i \not\Rightarrow W_y^i \xrightarrow{\text{p}} Rel_x^i$.

Therefore, in the example of Figure 3.1b, in the event of a crash, it may be the case that the release of Thread T0, which links a new node into the linked list, has persisted, but the preceding writes that created the node, have not persisted. This would leave the linked list in an inconsistent, and thus unrecoverable state.

3.3.2 ARP implementation

ARP is implemented on top of RCBSP [129], a persist-buffer-based BEP model. ARP modifies the ISA, enhancing the release and acquire instructions with persist semantics that enforce the ARP-rule. Note that for the ARP-rule, releases and acquires need not be treated as persist barriers: writes that precede a release need not be ordered with writes that follow the release and writes that follow an acquire need not be ordered with writes that precede the acquire. ARP enhances RCBSP to leverage this observation.

Recall that RCBSP orders all writes in the persist buffer (described in §3.2.2.1). On executing a persist barrier, the buffer's epoch is incremented such that subsequent writes belong to a later epoch, than writes that precede the barrier. ARP enhances this implementation as follows: on a release, no barrier is placed; rather a flag is raised denoting that the next acquire must place a persist barrier. On an acquire, a persist barrier is placed only if the flag is found raised. These additions enforce the ARP-rule as

follows: if a release Rel_x^i is inserted in the queue before an acquire Acq_y^j, then any write that precedes Rel_x^i must belong to an older epoch than any write that follows Acq_y^j, thus ensuring that writes that precede a release persist before writes that follow an acquire.

3.3.2.1 ARP implementation shortcomings

Firstly, we note that the ARP implementation abides by the ARP semantics, enforcing only the ARP-rule, without mirroring the RC orderings. For instance, a write that precedes a release is likely to belong to the same epoch as the release, and can thus persist after the release.

Secondly, even though the ARP authors identify that maximizing performance hinges on providing one-way persist barriers, their implementation still uses full persist barriers (i.e., not one-way). The lack of one-way barriers in the implementation makes it impossible to parallel the RC semantics: on the one hand, when the barrier is elided (i.e., on a release) APR fails to match the RC semantics, while on the other hand, when the barrier is placed (i.e., on an acquire) ARP provides more orderings than RC, as RC treats acquires as one-way barriers.

3.3.3 Why not simply fix ARP?

It is possible for ARP to honour the RC semantics, and thus enable the null recovery of log-free data structures, as long as a persist barrier is placed before every release. For instance, in the linked list example of Figure 3.1d, a persist barrier is also placed before the release, guaranteeing that the new node is persisted, before it can be linked to the structure.

Recall, however, that the design goal is not only to enable null recovery of LFDs, but also to maximize performance through one-way persist barrier semantics. By placing a persist barrier before every release, ARP regresses into a generic BEP model with full persist barriers. Aggravating the problem, the persist-buffer-based implementation of ARP pertains solely to full persist barriers, making it impossible to provide the desired one-way persist semantics.

Therefore, it is clear that there is a need for a new persistency model built from the ground up to provide efficient null recovery for log-free programs, by mirroring RC semantics through the use of one-way persist barriers.

3.4 Release Persistency

In this section, we introduce Release Persistency (RP), a persistency model that reconciles the performance of one-sided persist barriers with the stronger semantics that are required for the recovery of LFDs. First, we formally specify RP (§3.4.1), and then we discuss the performance implications of our specification (§3.4.2).

3.4.1 Formal Specification

RP must ensure that the persist order reflects the RC happens-before order, which we formally specified in §3.2.1, for enabling crash recovery. Note that because the persist order defines the order in which writes persist, only those RC rules that pertain to writes must translate into the RP formalism. Therefore, RP can be succinctly formalized as follows. Any two writes in the RC happens-before order must also persist in that order: $W1^i_x \xrightarrow{hb} W2^j_y \Rightarrow W1^i_x \xrightarrow{p} W2^j_y$

From the happens-before rules of §3.2.1, we can expand and specify RP via the following rules:

≻ **Release one-sided barrier semantics.** A write that precedes a release in program order appears before the release in persist order: $W^i_x \xrightarrow{\text{po}} Rel^i_y \Rightarrow W^i_x \xrightarrow{\text{p}} Rel^i_y$.

≻ **Acquire one-sided barrier semantics.** A write that follows an acquire in program order appears after the acquire in persist order: $Acq^i_y \xrightarrow{\text{po}} W^i_x \Rightarrow Acq^i_y \xrightarrow{\text{p}} W^i_x$.

≻ **Release synchronizes with acquire.** A release that synchronizes with an acquire appears before the acquire in persist order: $Rel^i_y \xrightarrow{\text{sw}} Acq^j_y \Rightarrow Rel^i_y \xrightarrow{\text{p}} Acq^j_y$.

≻ **Program order address dependency.** Two writes to the same address ordered in program order preserve their ordering in persist order: $W1^i_x \xrightarrow{\text{po}} W2^i_x \Rightarrow W1^i_x \xrightarrow{\text{p}} W2^i_x$.

≻ **RMW-atomicity axiom.** Read and write of an RMW appear consecutively in persist order: $R^i_x \xrightarrow{RMW} W^i_x \Rightarrow R^i_x \xrightarrow{p} W^i_x$ and there is no write W^j_y (from any thread) such that $R^i_x \xrightarrow{\text{p}} W^j_y \xrightarrow{\text{p}} W^i_x$.

A note on RP acquires. Because an acquire being a read cannot persist, the $Acq^i_y \xrightarrow{\text{p}} W^i_x$ ordering may appear bizarre at first. The intention here is to allow for two or more rules linked by an acquire to apply transitively.

For example, when a release synchronizes with an acquire, the released value must persist before any of the writes following the acquire persist. This is captured by applying the “release synchronizes with acquire” rule and the “acquire one-sided barrier

semantics" rule transitively.

In a similar vein, when a release synchronizes with an RMW marked acquire: (1) the released value must first persist; (2) then the value written by the RMW must persist (follows from the RMW atomicity axiom that mandates that read and write must appear consecutively in persist order); and finally (3) writes following the RMW must persist.

3.4.2 Specification implications

In Section 3.2.2.1, we discussed the conflicts that can occur in a cache-based implementation of a buffered epoch persistency model. Conflicts can adversely impact performance because they can trigger the persist of entire epochs in the critical path of one instruction's execution. However, not all types of conflicts are necessary to to capture the intention of RC; a number of these conflicts are merely an artifact of full persist barriers, which inescapably overshoot the necessary guarantees. One-way persist barriers capture the exact intention of RC and as a result substantially reduce the number of conflicts that need to be handled, pruning all unnecessary constraints.

Eliminated conflicts. Specifically, one-way persist barriers allow for a write to persist before writes of previous epochs. For example, assume $W1^i_x \xrightarrow{\text{po}} Rel^i_y \xrightarrow{\text{po}} W2^i_z$; even though $W2^i_z$ belongs to a later epoch than $W1^i_x$, the persist of $W2^i_z$ does not trigger the persist of $W1^i_x$. This relaxation substantially reduces the number of both inter- and intra-thread conflicts. Namely, examples of conflicts that are eliminated include: conflicts due to writing to a cache-line with an older epoch, conflicts triggered by evicting a cache-line and conflicts due to forwarding a cache-line upon receiving a coherence request. Notably, eliminating all these classes of conflicts allows for significant coalescing of writes to the same cache-line, which reduces the absolute number of persists.

Performance Implication. The ability to recover a program incurs an overhead in the program's execution, as its writes need to persist in the order mandated by the persistency model. The performance degrades because in order to honour the persist order, the processor often needs to stall. Note that, in eliminating the above types of conflicts, RP reduces the number of times the processor must stall. As a result, we hypothesize that the specification of RP can have a profound impact in performance. We prove this hypothesis in our evaluation section (§4.5).

3.5 Lazy Release Persistency

In this section, we present lazy release persistency (LRP), our microarchitectural mechanism for enforcing RP. As discussed in §3.2.2.1, a buffered implementation where visibility does not wait for persistency is critical for minimizing persistency-related overheads (also confirmed in our evaluation). This calls for a mechanism that maximizes buffering. Yet, the requirements of RP mandate the enforcement of inter-thread persist orderings (when a release synchronizes with an acquire). Allowing for buffering to extend across threads, however, incurs complexity in the form of coordination across memory controllers to enforce inter-thread persist dependencies correctly. It also involves complex deadlock-avoidance mechanisms to eliminate potential cyclic inter-thread dependencies [117][1]. Thus, we make the design choice of buffering persists within a thread until there is an inter-thread dependency, in which case we persist the buffered writes. Our evaluation (§3.6.4) vindicates this choice.

How to realize LRP with these design requirements? Our insight here is that the requirements matches that of lazy release consistency (LRC) [123], a protocol first proposed in the context of DSMs for enforcing RC lazily. Taking inspiration from LRC, we propose LRP a protocol for enforcing RP and its one-sided barriers. Next, we provide a high-level overview of LRP and how it satisfies the RP semantics (§3.5.1). We then dive into the specifics of our implementation (§3.5.2).

3.5.1 LRP Overview

In this section, we provide a high-level overview of LRP. We do this abstractly by describing the invariants satisfied by LRP. We informally argue for correctness by reasoning that the invariants are sufficient to enforce RP. In the next section, we discuss the detailed microarchitecture, explaining how LRP enforces the invariants.

LRP uses a buffering approach where persistency trails visibility. Therefore, writes to the L1 do not trigger persists. Instead, whenever a dirty cache-line is written back from the L1 (owing to eviction or a downgrade), the cache-line is persisted by the LLC/directory controller. LRP ensures that these persists enforce RP via ensuring four key invariants:

- **Invariant-1 (I1):** When the L1 controller receives an eviction request for a cache-line written by a release, it blocks the request until all of the cache-lines written

[1]Although DRF programs do not pose a deadlock risk, the hardware must be able to handle racy programs as well

by writes prior to the release have been persisted.

- **Invariant-2 (I2):** When the L1 controller receives a downgrade request for a cache-line written by a release from the directory, the request is blocked until: (a) all of the cache-lines written by writes prior to the release have been persisted; (b) the release has been persisted.

- **Invariant-3 (I3):** When an RMW, marked acquire, is successful (i.e., if the write is successful), the acquire blocks the pipeline until the write of the RMW persists.

- **Invariant-4 (I4):** When the directory controller receives a write-back from the L1, the directory persists the cache-line, blocking requests for the cache-line until it persists.

We now argue that the four invariants are sufficient to enforce RP's persistency rules.

$\succ$ **Release one-sided barrier semantics.** Invariant-1 ensures that before a release is allowed to persist, all previous writes have been persisted.

$\succ$ **Release synchronizes with acquire.** Suppose a release from thread T1 synchronizes with an acquire from thread T2 and issues a read request for the cache-line. There are three cases.

- Case-1: The acquired cache-line is in M state in T1's L1. In this case, T2's acquire would cause a coherence request (downgrade) to be sent to T1. Invariant-2 ensures that the acquire will block until the release and its preceding writes have been persisted, thereby ensuring this rule.

- Case-2: The acquired cache-line is in the LLC. Invariant-1 ensures that all writes before the release would have persisted. Invariant-4 ensures that the release itself would have persisted.

- Case-3: The acquired cache-line is in NVM. This implies that the release has already persisted. Invariant-1 ensures that all writes before the release also would have persisted.

$\succ$ **Acquire one-sided barrier semantics.** This comes naturally out of the consistency model. Any store following the acquire cannot perform (and hence cannot persist) before the acquire performs.

$\succ$ **Program order address dependency.** This again comes naturally. Since writes coalesce, it is impossible for two writes to the same variable to be inverted.

$\succ$ **RMW-atomicity axiom.** The only interesting case is when the RMW is marked an acquire. Invariant-3 ensures that the write of the RMW persists before following writes, thereby ensuring this rule.

An example. Consider the required semantics of Figure 3.1d: T0's W1 must persist before T0's Rel persists, and T0's Rel must persist before T1's W4. RP fulfills the requirements as follows. Let us assume that when T1's Acq performs the cache-line is held in T0's L1. Therefore, the Acq will trigger a downgrade request for the block, and hence from Invariant-2, T1's Acq will complete only after triggering the persist of T0's Rel and its previous writes (W1). Finally, T1's W4 cannot be issued to the memory system until T1's Acq completes, thereby ensuring $W1 \xrightarrow{p} Rel \xrightarrow{p} W4$.

3.5.2 LRP: Microarchitecture

We have established that LRP enforces the RP rules by upholding four invariants (I1-I4). I1 and I2 pose a significant microarchitectural challenge: on evicting/downgrading a released cache-line, all prior writes must be tracked and persisted. Conversely, I3 is trivially implemented by altering the processor pipeline to wait for an ack from the NVM controller on an RMW-acquire. I4 requires a minor alteration in the directory controller which we discuss more elaborately in §3.5.2.3.

Therefore, this section focuses on I1 and I2, presenting the a mechanism that, upon evicting/downgrading a release can scan the L1 cache and persist all prior writes. We note that the mechanism does not extend beyond the L1 cache and thus it can be simply implemented by enhancing the L1 controller, the L1 cache and the processor core. We begin by discussing the required hardware extensions.

3.5.2.1 Hardware extensions

Figure 3.3 illustrates all LRP hardware extensions, which are described one by one below.

Per thread metadata. Each (hardware) thread maintains an *epoch-id* counter which gets incremented on every release. In addition, the number of pending persists are denoted by a *pending-persists* counter. Upon issuing a persist for *any* write, the pending-

persists counter gets incremented; upon receiving an acknowledgment from the NVM controller for *any* of the issued persists, the pending-persists counter gets decremented. The pending-persists counter allows a persisting release to ensure that all previous writes have persisted.

Per L1-cache-line metadata. Each cache-line maintains: (1) a *min-epoch*, that holds the epoch of the earliest write to the cache-line and (2) a *release-bit*, that denotes whether the cache-line holds a value written by a release.

Release Epoch Table (RET). A small content-addressable table, called *Release Epoch Table* (RET), holds the *release-epoch* of cache-lines that hold a value written by a release. We note that it is possible to maintain a release-epoch for every L1 cache-line. However, we expect that at any given moment only a handful of cache-lines will hold values written by a release. This is because in most programs variables that are released account for a small percentage of the program's working dataset. Through our experiments, we have found that a 32-entry RET for each L1 cache adequately overprovisions for the needs of most programs. On executing a release, a RET entry is allocated, storing the release-epoch and the cache-line's address. When a release persists its respective entry in the RET is squashed. To avoid filling the RET, when the capacity reaches a watermark, the persist of the oldest release in the RET is triggered.

Persist engine. The *persist engine* is an FSM that takes as an input a release-epoch e_{rel} and scans the L1 cache, examining all cache-lines and persisting every cache-line with a smaller min-epoch than e_{rel}.

Hardware Overhead. We assume 32KB L1 cache with 40-bit tags. LRP adds an 8-bit min-epoch and a release-bit to each cache-line; amounting to 576 bytes for the entire L1. Note that when the epoch-id overflows, all not-yet-persisted cache-lines of L1 are persisted and the epochs are restarted. In addition, each of the 32 RET entries stores the physical address of a cache-line (i.e., 40 bits, same as the L1 tag) plus a release-epoch (8-bit), amounting to 192 bytes for the entire RET. In total, LRP requires less than 1KB per hardware thread.

3.5.2.2 A mechanism to enforce the release barrier

Having described the hardware extensions in each L1 controller, we now discuss how these extensions are leveraged to ensure that persisting a released cache-line triggers the persists of all writes of previous epochs. We then use this mechanism to enforce

invariants I1 and I2. But first, we establish some necessary terminology.

Terminology. If a cache-line holds a not-yet persisted write, the cache-line must reside in L1 in modified (M) coherence state. If a cache-line is in M state and has its release-bit set, it implies that the cache-line holds a value written by a release; we refer to such cache-lines as *released*. If the cache-line is in M state but its release-bit is not set, it implies that the cache-line holds a value written by regular writes only; we refer to such cache-lines as *only-written*. If a cache-line is neither released nor only-written, we refer to it as *clean*.

On a write. On performing a regular write, if the cache-line is clean, the thread's epoch-id is stored in the cache-line's min-epoch. If the cache-line is not clean, the write need not overwrite the cache-line's min-epoch, as the cache-line already has a valid min-epoch that is smaller than the current thread's epoch.

On a release. On a release, the thread's epoch-id is incremented; the new epoch will be the release-epoch, ensuring that all writes that precede the release are in an earlier epoch. There are two distinct cases for the state of the cache-line that the release intends to write. (1) Clean: the release assigns its epoch to the cache-line's min-epoch, it sets the cache-lines release-bit and it allocates a new entry in RET, where it also stores its release-epoch. (2) Not clean: the cache-line is first persisted and then treated as clean (i.e., case (1)). Note that case (2) implies that the release cannot be coalesced in the same cache-line with any previous write/release.

On a read/acquire. No additional action is necessary on a read or on an acquire.

On an RMW-acquire. An RMW marked acquire blocks the pipeline until its writes is persisted. Beyond this, additional action is not necessary.

On downgrading a cache-line. Attempting to downgrade a cache-line from M state triggers its persist. If the cache-line is only-written, then the persist happens off the critical path. But, if the cache-line is released, then the downgrade cannot complete before the cache-line has persisted.

On evicting a written/released cache-line. Evicting a cache-line that is written but not released, has the same effect as downgrading it. If the cache-line is released, then the eviction triggers its persist, but need not wait for the persist to complete (i.e., the persist is off the critical path).

Note that there is a subtle distinction between evicting and downgrading a released cache-line: while both actions cannot complete unless all previous writes/releases persist, downgrading also requires that the released cache-line itself persists; there is no

such requirement for evicting. To simplify the rest of the discussion, we refer to both downgrading and eviction as the act of persisting a released cache-line. To enforce the eviction invariant (i.e., I1) we do not wait for an ack from the NVM controller for the released cache-line, while to enforce the downgrade Invariant (i.e., I2), we wait for the ack.

On persisting a released cache-line. First, the RET is accessed to read out the release-epoch e_{rel} of the cache-line. The e_{rel} along with the address of the cache-line are propagated to the persist engine, which begins scanning the L1 cache, discovering all only-written/released cache-lines with min-epoch smaller than e_{rel}. The persist engine must issue a persist for all discovered cache lines, but there is a catch: amongst the discovered cache-lines, there may exist a released cache-line CL_r with epoch e_k and a written cache-line CL_w with epoch e_{k-1}, for which the release one-way barrier semantics mandate that CL_w must persist before CL_r.

Figure 3.2 illustrates this case. When attempting to persist the Release ($F2$), the persist engine tracks down all only-written/released cache-lines of previous epochs. One of the tracked cache-lines will be the released CL_c which holds the Release($F1$) and the only-written CL_dt which holds the Write(X). The one-way persist barrier semantics of the release mandate that the only-written CL_d must persist before the released CL_c.

Persist engine algorithm. The persist engine achieves this ordering by persisting first all the only-written cache-lines and then persisting the released cache-lines in their epoch order. Specifically, the persist engine operates as follows: as the persist engine scans the L1 cache it keeps discovering cache-lines that must be persisted; on discovering a only-written cache-line, it immediately schedules its persist, incrementing the pending-persists counter. Otherwise, on discovering a released cache-line, it simply buffers it in a local queue inside the persist engine. In either case, the engine immediately resumes scanning the L1 cache.

After the scanning completes, the engine starts polling on the pending-persist counter, waiting for it to become zero. Recall, that the pending-persist counter gets decremented every time an ack from the memory controller reaches the L1, denoting that a pending persist has completed. When the pending-persist counter reaches zero, the persist engine infers that all scheduled persists have completed, and thus it can start scheduling the persists of the released cache-lines.

For instance, in the example of Figure 3.2, the persist engine first persists the written cache-lines Cl_a, CL_b, and CL_d; then, and only after these cache-lines have persisted,

the persist engine will first persist CL_c that holds Release(F1) and then CL_e that holds Release(F2).

Enforcing invariants I1 and I2. The persist engine algorithm enforces both I1 and I2 by persisting a release, with one simple distinction: for I1 (i.e., release eviction) the persist engine does not wait for the released cache-line to be acked by the NVM memory controller, while for I2 (i.e., release downgrade) it waits for the released cache-line to be acked.

Persist engine correctness. The persist engine essentially reorders the persist of writes with the persist of prior releases, while ensuring that releases persist in their epoch order. This reordering abides by the RC happens-before order, because a release is a one-way barrier for prior accesses, thus allowing the reordering with subsequent accesses.

3.5.2.3 Coherence controller

LRP involved modest (local) changes to the L1 coherence controller. Specifically, a downgrade request (e.g. a Fwd-GetS request) for a released cache-line in M state could block until previous writes in the L1 (if any) persist. It is important to note that this does not pose a deadlock risk since the persist actions are guaranteed to complete without themselves being blocked.

Thus far, we have discussed a stalling implementation where the cache controller blocks on a Fwd-GetS. Stalling is not fundamental to our technique. It is possible to avoid this stalling by moving to a transient state upon a Fwd-GetS, which logically moves the state to S, waiting on an acknowledgement from the persist engine to move back to a stable state. However, we have not experimented with this non-stalling variant yet.

LRP also involves a minor change to the directory controller. Upon an L1 eviction of a released cache-line, a PutM request is sent to the directory. Normally, the directory would immediately transition to S state. However, the directory now enters a transient state that would block coherence requests to (only) that block until it receives a persist acknowledgement. Note that this does not stall the directory controller and hence is not expected to affect its performance.

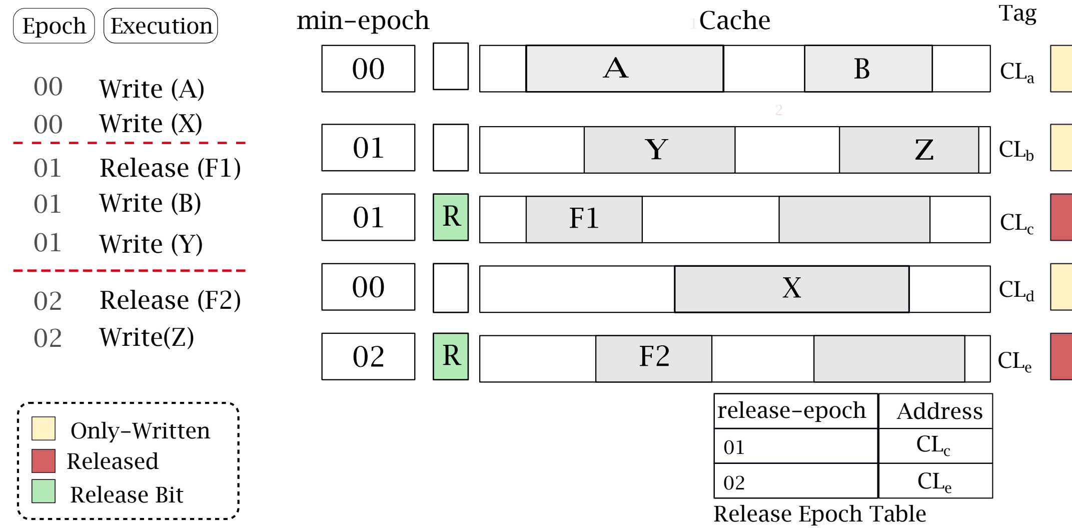


Figure 3.2: The state of the cache after an example execution.

3.6 Experimental Evaluation

Thus far, we have established that RP must be enforced for enabling recovery of LFDs. We conducted experiments seeking to answer two main questions. First and foremost, how much does our one-sided barrier mechanism (LRP) improve on the state-of-the-art full barrier when enforcing RP? Second, how much performance overhead does enforcing RP incur over a volatile execution that provides no persistency guarantees? Before we go to the results, we first discuss our workloads and methodology.

3.6.1 Workloads

LFDs are essentially nonblocking data structures with persist barriers inserted for ensuring crash recovery. We obtained 4 of our workloads from the SynchroBench suite [92], which is a collection of nonblocking data structures. Specifically, we used the linkedlist [101], hashtable [156], binary search tree (balanced tree) [167] and skip-list [218] workloads. We also implemented the lock-free queue from Michael and Scott [157]. All workloads are data-race-free in that synchronization operations are properly labelled using releases and acquires. For each workload, we use a harness that creates 1–32 workers and issues inserts and deletes at 1:1 ratio. Since we only use insert and delete operations, the *update-rate* of the benchmark suite is 100%. The data structure size refers to the initial number of nodes in the data structure before statistics are collected: we vary the size from 8K entries–1M entries, and the default value is 64K

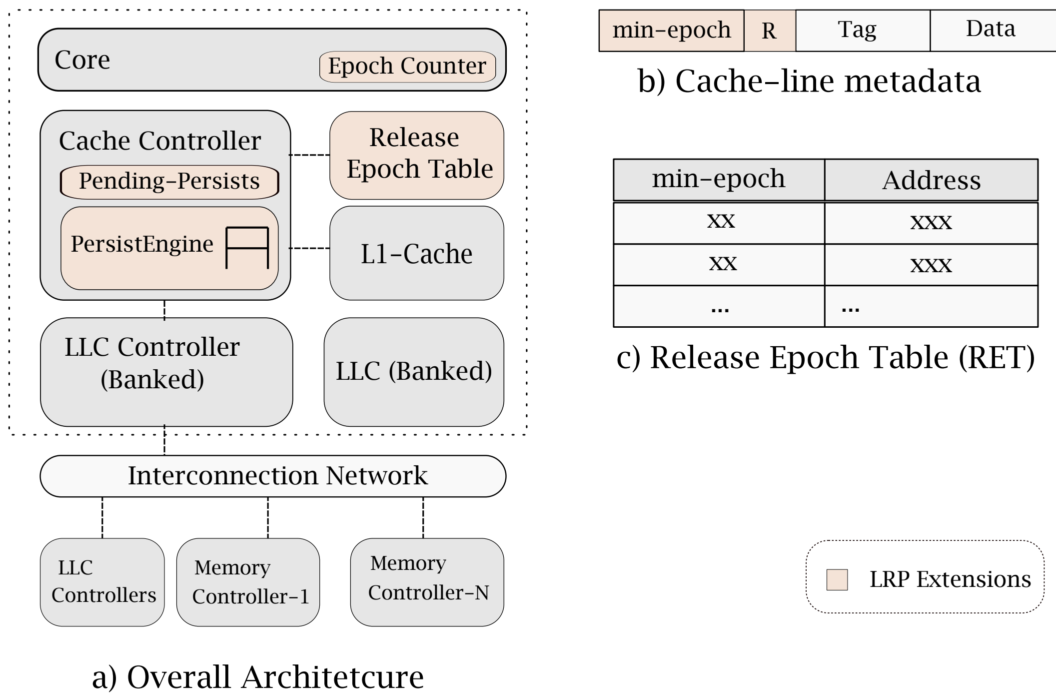


Figure 3.3: Hardware extensions involved in LRP

entries.

3.6.2 Comparison Points

We compare LRP against alternative methods for enforcing RP using full barriers. We also compare against volatile execution.

LRP. This represents our approach for enforcing RP. Releases and acquires are automatically treated as one-sided barriers and perform the actions described in §3.5.

SB. This represents an RP enforcement approach using a strict full barrier (SB). Recall that SB blocks until all the cache lines modified by the writes before the barrier has persisted. SB also has an inter-thread component; when a shared memory dependency is detected via the coherence protocol, the target thread blocks until the writes in the ongoing epoch of the source thread have persisted. Therefore, in order to enforce RP: (1) an SB has to be inserted before each release to ensure that all writes before the release persist before the release; (2) an SB also has to be inserted after the release to ensure that inter-thread persist ordering is captured. (i.e., to ensure that when a release synchronizes with an acquire, the acquire correctly blocks until the release persists.)

BB. This represents an RP enforcement approach using the the state-of-the-art full

barrier [117]. As discussed in §4.2, the barrier enforces the persist orderings (both intra-thread and inter-thread) similarly to SB, but employs an efficient buffered implementation that minimizes blocking. Hence, we refer to it as buffered barrier (BB). In order to enforce RP: (1) a BB has to be inserted before each release to ensure that all writes before the release persist before the release; (2) a BB has to be inserted after the release and before the acquire for capturing the inter-thread persist ordering (i.e., to ensure that when a release synchronizes with an acquire, all writes following the acquire should persist after the release persists).

NOP. Finally, we also compare against volatile execution which does not enforce any persistency model (NOP).

Processor	64-Core (Out-of-Order) 2.5 GHz
ISA	Intel x86-64
L1 I+D -Cache (pvt.) line-width	32KB, 2 cycles, 8-way 64B
L2 (NUCA, shared)	1MB x64 tiles, 16-way 15 cycles
On-chip Network	2D-Mesh 32 bit flits and links
Coherence	Directory-based, MESI
NVM (PCM)	cached mode: 120 cycles uncached mode: 350 cycles
RET (private)	32 Entries

Table 3.1: Simulator Configuration

3.6.3 Simulator

Our hardware implementation is built on top of the pin-based [150] PRiME [75] simulator, with 64 in-order-cores processor (single thread per core), a logically shared LLC and multiple memory controllers. Table 3.1 shows the details of the simulated processor and memory system.

We model NVM latencies based on the performance measurements observed on Intel Optane persistent memory [113]. Specifically, there are two modes that determine the NVM latency. In the *cached mode*, an NVM writeback persists as soon as it is

written to a battery-backed NVM-side DRAM cache. In the uncached mode, an NVM writeback persists only after it is actually written to the NVM. We assume the faster cached mode for our experiments unless specified otherwise.

PRiME only supports x86-64 ISA and hence enforces the TSO (Total-Store-Order) consistency model. As such the simulator lacks releases and acquires in its ISA. Therefore, we implemented a simple extension to the ISA for taking in release/acquire annotations. We make use of Pin's capability to instrument the binary and generate these special stores and loads with release/acquire annotations corresponding to releases and acquires in the program.

It is worth noting that we did not alter the simulator's consistency enforcement mechanism to take advantage of the release/acquire annotations. (This is sound because TSO stores and loads already have release and acquire semantics respectively.) However, we take advantage of these annotations to implement our LRP mechanisms in order to enforce RP.

3.6.4 Results

LRP outperforms BB and SB. Figure 3.4 shows the execution times of LRP, BB, and SB normalized to NOP with 32 worker threads and 64K elements. We first observe that BB outperforms SP, showing a 26%-68% (average 52%) improvement over BB. This is primarily because BB, which is a buffered implementation, avoids stalls in the critical path. This vindicates our design decision of striving for a buffered implementation for enforcing RP. How does LRP stack up against BB? Our key result is that LRP significantly outperforms BB, showing a 16%-55% (average 33%) improvement over BB.

LRP is 5%-12% within NOP. Figure 3.4 also reveals that LRP is only 5%-12% (average 9%) within volatile execution which suggests that the persistency-related overheads incurred by RP is nominal for these workloads.

Why LRP outperforms BB? Recall that the expected advantage of LRP over BB is that it significantly minimizes intra-thread persistency overheads being a one-sided barrier. On the other hand, BB is expected to incur lesser inter-thread persistency overhead; this is because, whereas LRP blocks on an acquire to enforce the inter-thread persistency orderings, BB enforces those lazily well. To understand why LRP outperforms BB, we conducted experiments to study the effect of intra- vs inter-thread persistency overheads of LRP and BB.

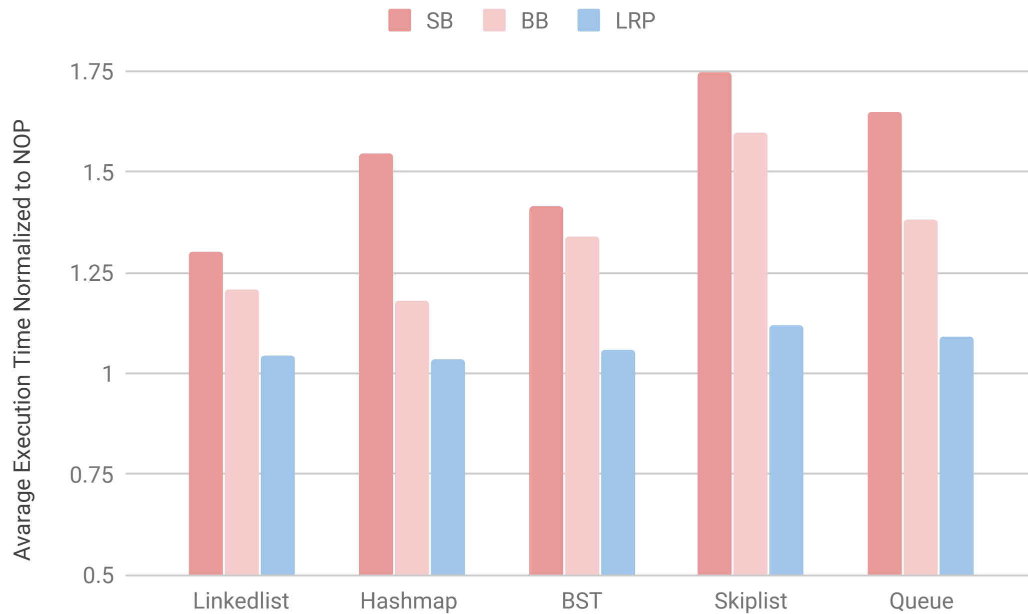


Figure 3.4: Execution time normalized to No-Persistency (lower the better).

In Figure 3.5, we classify write backs into two categories: those that are in the critical path of the execution (of the processor doing the write back) and those that are not. For BB, a significant 69% of the write backs are in the critical path, whereas for LRP only 15% of the write backs are in the critical path. Since almost all of the write back are due to persistency orderings, this suggests that LRP significantly minimizes intra-processing overheads in comparison with BB.

Figure 3.7 and Figure 3.8 compare the normalized execution time overheads of RP vs BB as the number of worker threads are varied from 1–32. The greater the number of threads, the greater the probability of inter-thread conflicts and hence potentially high inter-thread persist ordering overhead for RP. However, as seen in Figure 3.7 and Figure 3.8, this effect is nominal: for RP the persistency overhead remains relatively flat with increasing threads. For BB there is a marginal decrease in performance overhead as the number of threads is increased.

The above two experiments suggest that the effect of intra-thread persistency overhead far outweighs the effect of inter-thread persistency overhead. Therefore, this vindicates the design choice of RP in seeking to optimize away the intra-processing overheads vs inter-thread overheads.

Individual workload analysis. Whereas RP consistently outperforms BB, as we can see from Figure 3.4, the gap between RP and BP varies. One trend we observed is

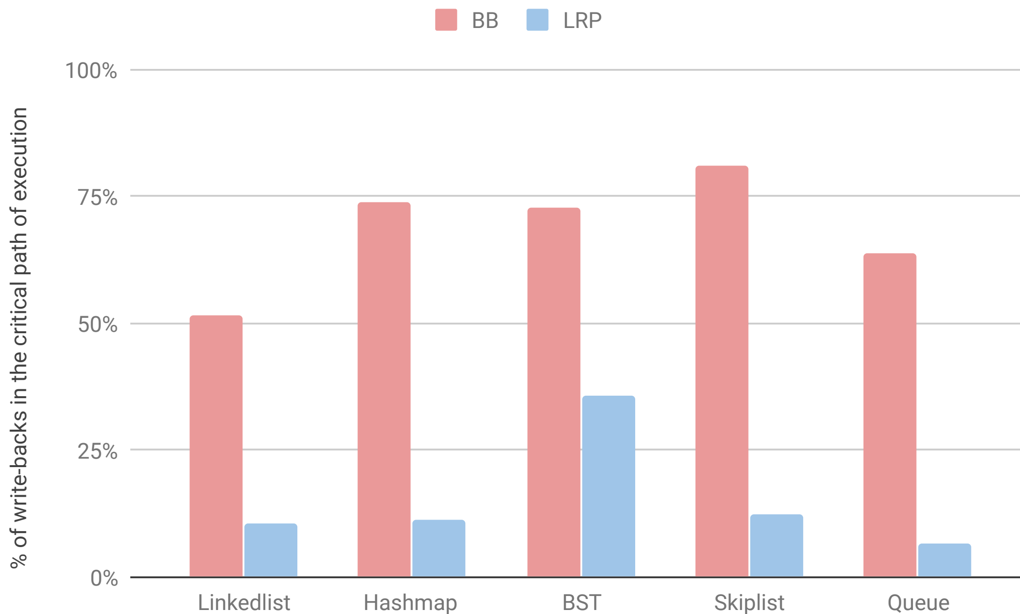


Figure 3.5: Percentage of write backs in the critical path

that, for read-intensive workloads, the gap between RP and BP is smaller than for write-intensive ones. As discussed earlier, BB suffers from intra-thread conflicts and these are more pronounced for write-intensive workloads. Thus, we can observe that linkedlist, a read-intensive workload owing to read-heavy link traversals, shows lesser gain over BB 19% gain compared to BST, a write-intensive workload which shows a relatively higher 37% gain.

Cached vs Uncached mode. Recall that up until now we assumed the cached mode where a write back is said to persist as soon as it reaches the NVM-side DRAM cache. In this experiment, we consider the uncached mode by disabling the NVM-side DRAM cache, thereby exposing the slower NVM to applications. Figure 3.6 presents the normalized execution time overhead over NOP on the uncached mode. As we can see, and comparing with the results on the cache mode shown in Figure 3.6, RP is more robust to this change when compared with BB or SB. RP continues to incur a nominal 6%-20% (average 11%) overhead compared to NOP. BB (and SB) are affected more by this change because they have more writebacks in the critical path when compared to RP. Thus, RP shows a significant 76% improvement over BB in this configuration.

Sensitivity to Data structure size. In order to measure the sensitivity of RP to data structure size, we varied the size from 8K–1M nodes. However, we did not observe a significant change in the results (and hence we do not show these results). Changing the

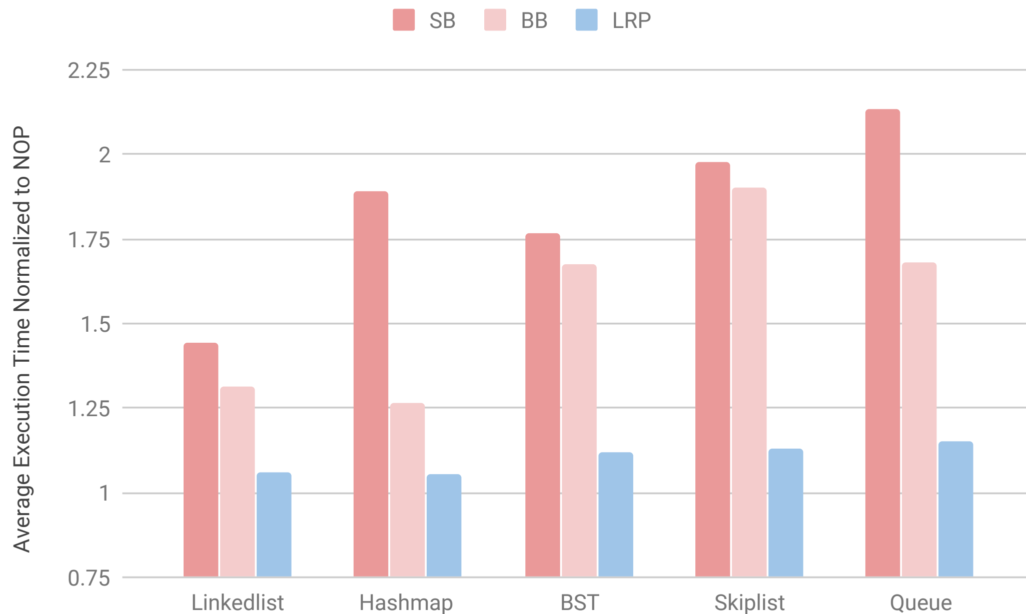


Figure 3.6: Execution time normalized to No-Persistency in the Uncached mode (lower the better).

number of elements in the data structure largely affects inter-thread conflicts compared to intra-thread. Our observation is that even though the number of inter-thread conflicts changes over the data structure size, it does not affect the execution time overheads significantly because, as established earlier, the effect of intra-thread conflicts is more significant.

3.7 Summary

We have argued that languages must support ordering primitives that are strong enough to enable recovery of log-free data structures (LFDs) without compromising on efficiency. Specifically, the release (and the writes before it) must persist before the writes following the acquire persists. We formalize this behavior via a persistency model called release persistency (RP). The challenge is to realize RP with one-sided barriers while also retaining a buffered implementation where visibility does not wait for persistency. I.e, we must strike a good balance between laziness and eagerness. We achieve this with a buffered implementation in the intra-processor sense yet one with persistency catches up with visibility when an acquire synchronizes with a release—detected and enforced by piggybacking on top of the coherence protocol. Our experiments on commonly used

LFDs suggest that our one-sided barriers efficiently enforce RP, significantly improving upon the state of the art.

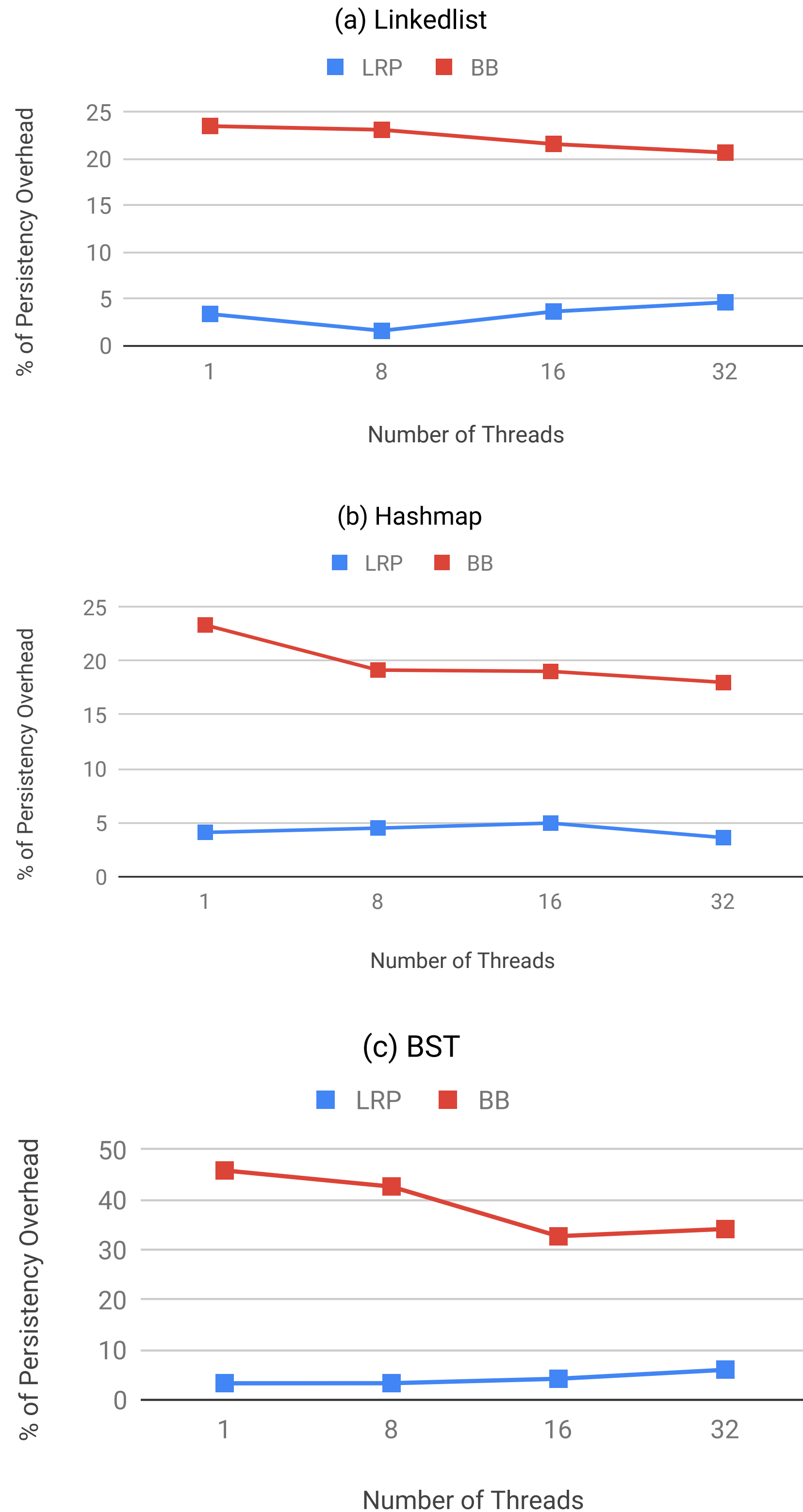


Figure 3.7: PART 1: Percentage overhead over and above No-Persistency, varying the number of threads from 1 through 32. (lower the better)

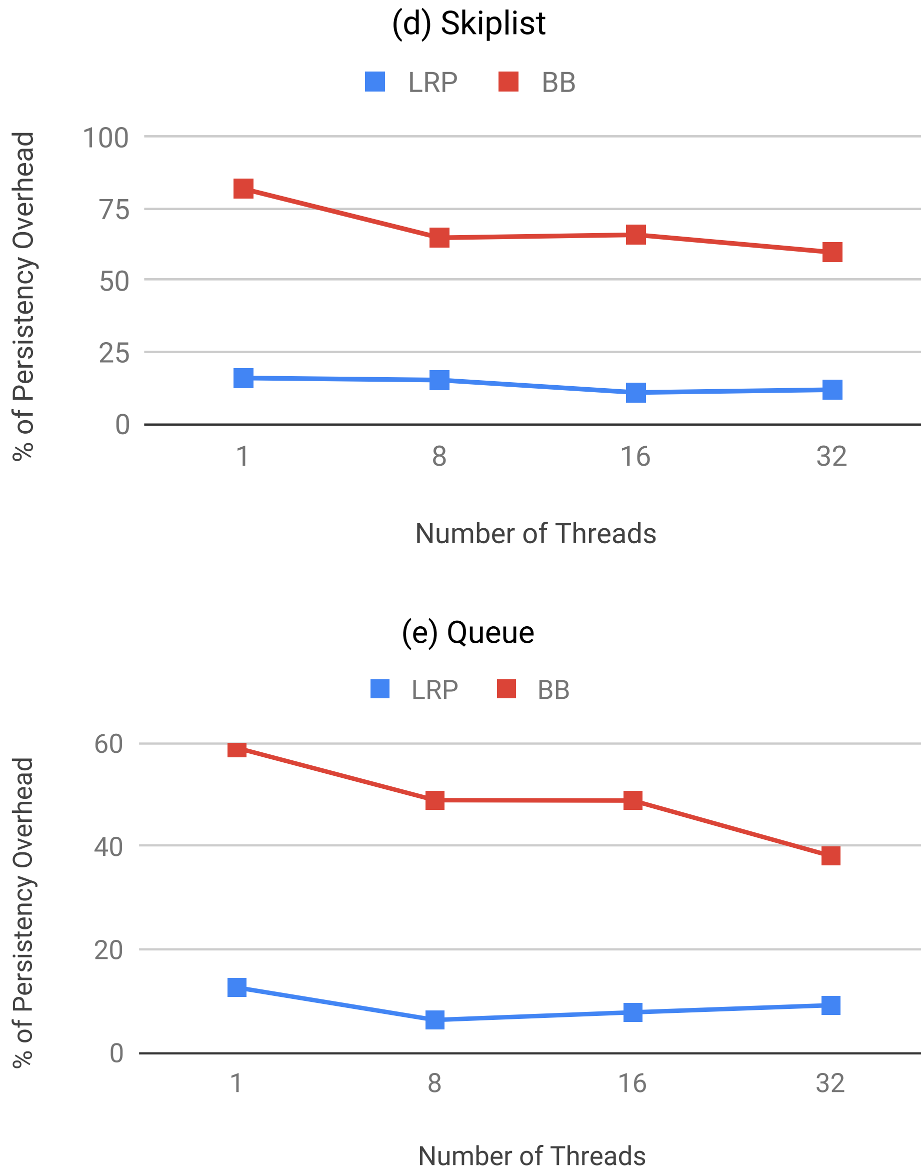


Figure 3.8: Part 2: Percentage overhead over and above No-Persistency, varying the number of threads from 1 through 32. (lower the better)

Chapter 4

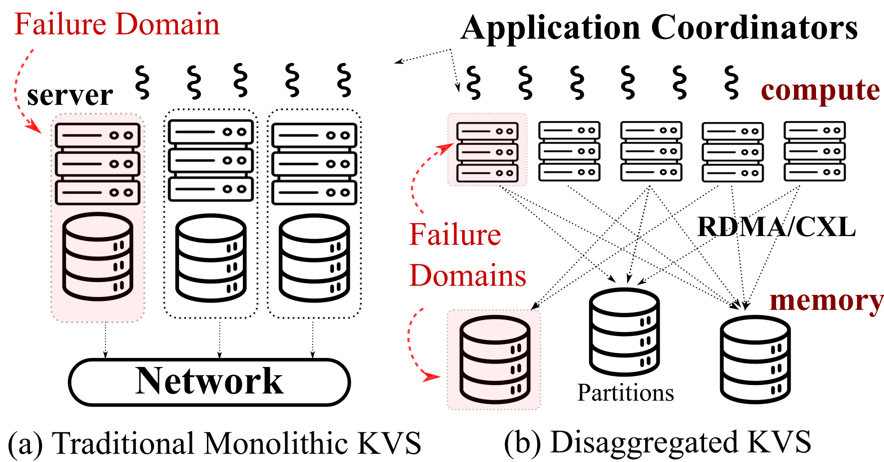
Pandora: Highly Available, Recoverable Transactions on Disaggregated Data Stores

4.1 Introduction

In this thesis, we explore the recovery of transactional, in-memory key-value stores over disaggregated memory (DM) architecture. We argue that existing work has not studied how to recover correctly and efficiently under disaggregated memory. To address that, we propose our own approach to tackling recovery in this new setting. But first, let us provide some context.

Both academia [190, 145, 99, 200, 136, 181, 91, 191, 130] and industry [47, 42, 138, 11, 10, 12] are exploring DM to mitigate the inefficiency caused by the fixed compute-to-memory ratio in traditional datacenter servers. This inefficiency arises when an application needs more compute than memory, or vice versa, but the server's fixed ratio doesn't match its requirements. DM decouples memory from compute by deploying two types of servers: 1) *memory servers* that provide ample memory with minimal compute and 2) *compute servers* that offer high compute capabilities with minimal memory. Memory servers are connected to compute servers through a high-speed RDMA network, which allows memory and compute to be efficiently provisioned according to the needs of the application.

The primary advantage of DM is improved resource utilization, but there is another crucial benefit. With memory and compute now operating independently, distributed applications can keep running even if a compute server fails, since no memory is lost



(a) Traditional Monolithic KVS (b) Disaggregated KVS

Figure 4.1: Independent failures in Disaggregated Memory. Because memory and compute are independent in DM, distributed applications could keep running even if a compute server fails.

(Figure 4.1). I.e. Recovery from compute failures can be *non-blocking*. This is in contrast to traditional monolithic server architecture, where a server failure results in the automatic loss of a portion of memory that acts as the primary storage for a set of keys.

This work leverages this observation in the context of transactional in-memory key-value stores (dubbed KVSes). Such KVSes are a crucial component of the datacenter infrastructure [59, 60, 126, 120], and their availability in the presence of failures is critical. When a KVS server fails, a portion of the objects becomes inaccessible. Although all objects are replicated, the entire KVS must stop briefly to – at least – reconfigure itself and steer requests for the inaccessible objects to other replicas. In KVSes the ratio between compute (transactions per second) and memory (dataset size) can vary arbitrarily across the numerous use cases. This makes them a perfect fit for DM. In DM-KVSes (dubbed DKVSes), the compute servers are responsible for coordinating transactions while memory servers hold the dataset passively. In this architecture, there is no reason why the failure of a compute server should cause any interruption to the operation of the DKVS.

Unfortunately, there is no existing DKVS that offers this capability. Providing this capability will require a new DM-based recovery protocol. The key challenge in designing this protocol is that compute servers can only access memory through the limited one-sided RDMA API (read, write, compare-and-swap, and fetch-and-add). This is in contrast to traditional, non-disaggregated architectures where servers can send arbitrary remote procedure calls (RPCs) to one another.

Over the past decade, researchers have studied this RPC-to-RDMA transformation in the context of transactional protocols [219, 158, 59, 126]. Specifically, FaRM [59, 60] showed how to implement the execution phase of a transactional protocol over one-sided RDMA without RPCs. FORD [219], which is the first and only existing (transactional) DKVS, took the next step, designing the entire transactional protocol (including the execution, validation and the commit/abort phases) solely through one-sided RDMA. However, FORD did not consider the challenge of efficiently recovering from a compute failure.

4.1.1 Pandora

In this thesis, we take the next step towards a correct, performant, and highly-available DKVS. We propose Pandora, a fully one-sided transactional protocol that ensures memory is always in a recoverable state and includes special handling of compute failures to avoid unnecessary interruption. Using FORD as the steady-state protocol, we design an RDMA-based recovery protocol that detects and recovers from a compute failure while eliminating interruption. To ensure correctness, we introduce an end-to-end litmus testing framework that revealed a number of bugs on FORD, which prohibited recovery from being fast, correct, or non-blocking. Finally, we integrate our innovations into the FORD system [219] and thoroughly validate and evaluate our proposal.

Implicit Latch Logging. One problem with recovering after a compute failure in FORD is that the FORD protocol latches[1] keys before logging. Thus, post-failure, the entire memory must be scanned to discover and undo any not-yet-logged latches taken by the failed compute server. This operation can take multiple seconds: e.g., scanning 100 GiBs through a 100Gbps network link will require at least 8 seconds. During this period, the rest of the compute servers must block. Crucially, to solve this, we cannot simply reorder logging and latching, because that will either require a heavy redesign of the protocol or impose overheads by adding extra messages. Instead, we propose a new RDMA-friendly technique called *Implicit Latch Logging*, where we extend the latch structure to also include the id of the compute server. When failing to latch, a compute server will inspect the latch to see if the current server holding the latch is a failed compute server. If so, the latch can be *stolen*.

RDMA-based Recovery protocol. To detect and handle compute failures, we propose

[1] Throughout this work, we say “latch” instead of “lock”.

an RDMA-based recovery protocol that works in four steps. First, it uses heartbeats to detect failures. Second, it revokes the RDMA rights of the failed server to ensure safety even under false positives. Third, it reads the logs of the failed compute server (which are stored in the memory servers) and either rolls forward or rolls back all of its logged transactions. Finally, it notifies the remaining compute servers of the failure so that they can acquire any stray latches of the failed server. Crucially, during this process, the alive compute servers can continue executing transactions.

End-to-end Litmus Testing. There are several factors that render the recovery protocol particularly error-prone. Firstly, recovery is a complicated distributed algorithm that must be able to detect failures and roll back and forward uncommitted transactions. The recovery is only executed once per failure, limiting the ability to uncover corner cases; contrast this with the rest of the transactional protocol, which is executed millions of times per second. Secondly, it does not suffice for only the recovery protocol to be correct; for recovery to work, the transactional protocol must ensure that memory is always in a recoverable state. However, this aspect of the protocol is not tested during failure-free operation and thus is very error-prone.

In this thesis, we introduce a new litmus-testing framework for end-to-end validation of transactional protocols in general, and Pandora in particular. Litmus tests are small transaction sequences that are designed to expose bugs. To the best of our knowledge, this is the first work to create litmus tests and a framework for deploying these tests for validating transactional protocols. Our validation revealed multiple subtle bugs in FORD, which can – in rare cases – leave the memory in an unrecoverable state, each of which has been fixed in Pandora.

4.1.2 Contributions

- We observe that disaggregation presents an opportunity for highly available (transactional) KVSes. However, the limited one-sided RDMA semantics (and the absence of traditional RPCs) poses a challenge to fast recovery (§4.2).

- We propose Pandora, a fully one-sided transaction and recovery protocol specifically designed to achieve correct and fast recoverable transactions on DKVSes.

- Pandora consists of two innovations: Implicit Latch Logging (§4.3.1), an RDMA-friendly technique for making latches recoverable in the presence of failures, and new RDMA-based recovery protocol (§4.3.2) for detecting and recovering from

failures.

- To validate correctness, we introduce a new litmus-testing framework (§5.1), and our validation reveals multiple subtle bugs in the FORD protocol, each of which is addressed in Pandora.

- We implement our techniques on top of the FORD system, the only state-of-the-art DKVS. Our evaluation shows that Pandora achieves fast recovery in the order of a few milliseconds during which live compute servers proceed with their transactions and while incurring nominal overhead on failure-free execution (§4.5).

4.2 Preliminaries

In this section, we first provide a brief background on disaggregated key-value-stores (DKVS). We then briefly discuss FORD, which serves as the basis for this work.

4.2.1 Disaggregated KVS (DKVS)

Researchers from academia and industry are advocating for the adoption of disaggregated memory (DM), arguing that it improves scalability, power utilization and cost efficiency [99, 145, 15, 47, 164, 205, 33, 217]. In a DM architecture, servers are divided into *compute* and *memory*. Compute servers have the compute capabilities of today's commodity servers, but limited memory (i.e., a few GiB) for caching but not in-memory storage. Memory servers have a lot of memory for storage but near-zero compute [200, 136, 219]. As in recent DM works [61, 207], we assume that memory servers have a small set of wimpy cores (1 - 2) to support lightweight connection management and initialization but do not traverse indexes or apply transactional logic. Instead, compute servers perform those over the memory servers through one-sided RDMA.

This work focuses on Key-Value Stores deployed over DM, or simply DKVSes. Specifically, we focus on DKVSes that replicate and distribute their data *in-memory* across multiple memory servers. A set of compute servers run the DKVS library, which offers a simple transactional API. Applications express their transactions through requests that include calls to BeginTx, Read, Write, Insert, Delete, and CommitTx. An application can run on the same servers as the DKVS library or on remote servers. In

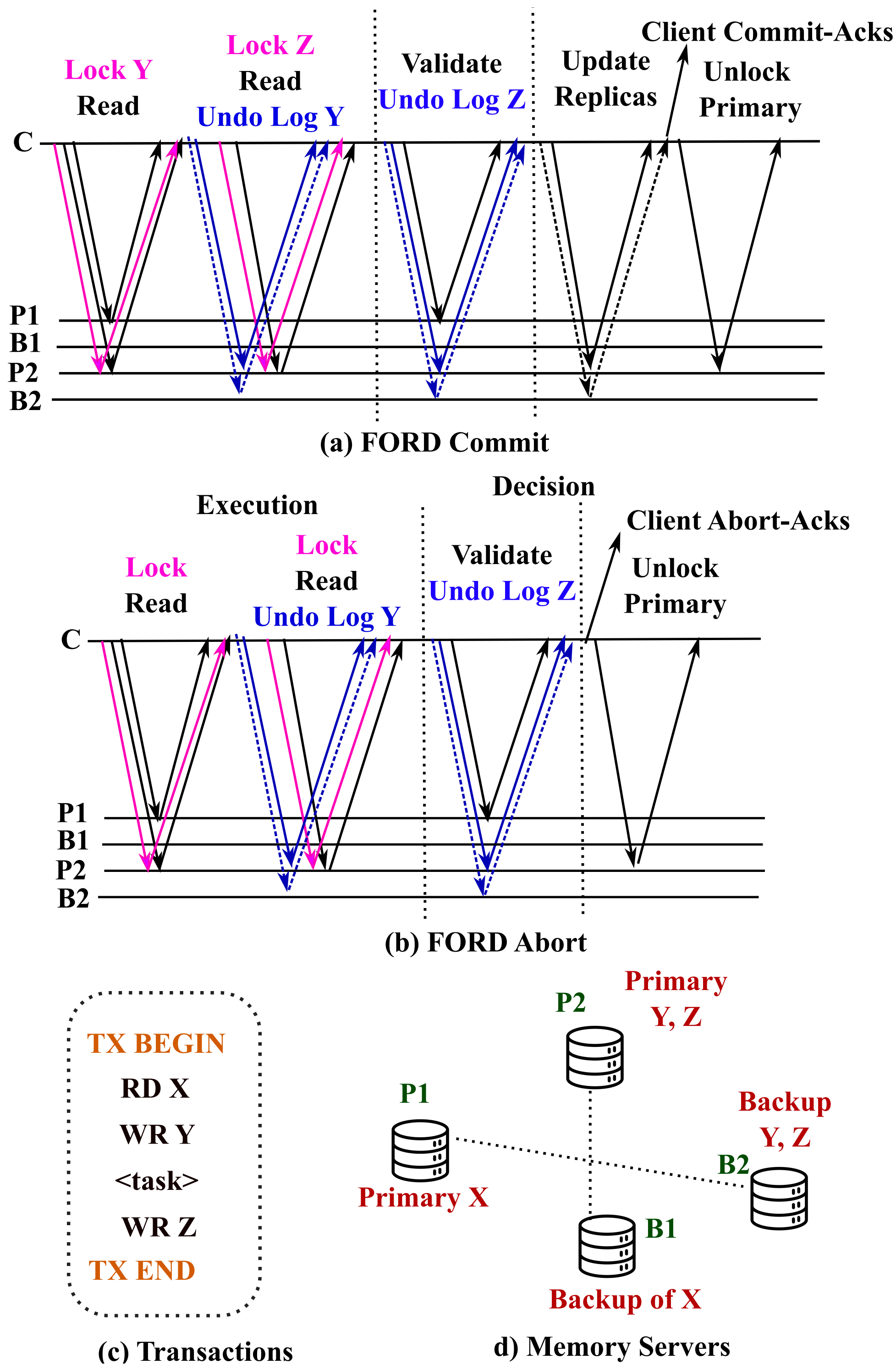


Figure 4.2: FORD's (a) commit path, and (b) abort path (only if any latch or read-validation fails before the decision); (c) example of a transaction that reads object X, writes object Y, before writing object Z; (d) X has P1 as primary while Y and Z have P2. P1 and P2 are replicated in B1 and B2. For instance, in (a) coordinator first reads X from P1, then latches and reads Y from P2 and performs a task locally before latching and reading Z from P2, then reads the version of X from P1 for read-validation while writing undo logs in P2 (and its backup B2) in the background; finally, it updates Y and Z in-place in P2 and B2 before unlatching Y and Z in P2. (b) is similar.

either case, the applications' requests are routed to the DKVS library, which executes a *transactional protocol*, accessing and replicating DKVS data as needed.

Architecture. The compute server responsible for executing the protocol for a transaction is referred to as its *coordinator*. Each object is stored in multiple memory servers. Every object is assigned a *primary* memory server, with the remaining servers designated as *backups*. An object can only be accessed through its primary. The backups are kept consistent with the primary so that they can take over in the event of a failure.

Consistency and Failure Model. As in prior works in distributed replicated transactions [59, 219, 120, 209], we focus on transactions that provide the strongest consistency guarantee (i.e., *strictly serializability* [188]). We consider a non-byzantine partially synchronous model [64] with crash-stop compute and (up to $f+1$) memory server failures as well as network faults, including message reordering, duplication, and loss.

4.2.2 Recoverable Transaction Protocol

A recoverable transactional protocol is responsible for ensuring consistency (i.e., strict serializability) and handling failures under the aforementioned failure model. We find it useful to classify the actions of the protocol under three categories.

C1. Online-failure-free. This includes all of the actions required to ensure transaction correctness (i.e., strict serializability) *when there are no faults*.

C2. Online-recovery. This includes the actions from the protocol responsible for *maintaining the state required to facilitate recovery should a failure happen*. Typically, this includes all of the steps involved in logging data and metadata.

C3. Recovery. This includes all of the actions from the protocol for detecting and recovering from a compute server failure. More specifically, it includes the actions for detecting the failure and ensuring that the failed server cannot affect the system anymore (e.g., in case of a false positive). It also includes actions that ensure that transactions from the failed server have either been rolled back or rolled forward completely.

Key features of a recoverable transactional protocol. An ideal recoverable protocol should have four key features.

1. **Correctness.** First and foremost, the protocol must ensure correctness (i.e., strict serializability) both in the absence and presence of compute failures.

2. **Minimal online overhead.** The online-recovery component of the protocol

should ideally add as little overhead as possible to the online-failure-free component; i.e., the overhead of logging should be as minimal as possible.

3. **Fast recovery.** The recovery component should be fast; i.e., it must quickly roll back or roll forward any pending transactions on the failed compute server, so that these transactions (and any other transactions from other compute servers conflicting with the above transactions) can make progress as soon as possible.

4. **Non-blocking.** Recovery must not block other non-conflicting transactions from other computer servers; these transactions must continue to make forward progress despite the computer server failure.

The combination of the non-blocking property (for non-conflicting transactions) and fast recovery (which affects the latency of conflicting transactions) is what makes transactions highly available.

4.2.3 FORD

This section presents FORD [219], the only published DKVS to date. FORD executes distributed transactions using a variant of the OCC transactional protocol [131], which offers strict serializability [189]. Specifically, FORD's protocol consists of three phases: execution, validation and commit/abort.

1. Execution. During execution, the coordinator reads all objects in its read-set, and reads and latches all objects in its write-set. The execution phase will fail if any accessed object is already latched. If execution succeeds, the protocol moves to validation. Otherwise, it moves to the abort phase.

2. Validation. For validation, the coordinator checks that all objects in its read-set are still in the same state, i.e. they have the same version and have not been latched. This ensures that the transaction is working over a consistent view. When the validation phase completes, then we say that we have reached a *decision*: the transaction will either commit or abort.

3. Commit/Abort. Commit entails two steps: 1) all writes are applied to both the primary and backups of each object and 2) latched objects are unlatched. The client is notified after the first step with either a *commit-ack* or an *abort-ack*. Conversely, to abort, we simply unlatch all latched objects and then notify the client.

Undo Logging. During the first two phases, the protocol writes an undo log in the

primary and every backup of every object in its write-set. The purpose of this is to facilitate recovery in the event of a failure.

Figure 4.2 illustrates FORD. Figure 4.2(a) and (b) show the commit and abort path of the transaction shown in Figure 4.2(c) which reads object X and writes object Y. Figure 4.2(d) shows the four assumed memory servers, Each of which serves as primary or backup for objects X and Y.

FORD summary. Going back to our classification, the execution, validation, and commit/abort phases of the protocol comprise the online-failure-free (C1) component of FORD. The Undo logging comprises the online-recovery component (C2).

In the next section, we will discuss the limitations of FORD's logging component in the presence of compute server failures, and how it is addressed in Pandora. We will also delve into Pandora's recovery algorithm, which addresses the lack of a recovery component in FORD.

4.3 Pandora

In this section, we discuss Pandora, a highly-available transactional protocol that recovers efficiently on a compute failure. Recall that in Section 4.2, we split a transactional protocol into three distinct categories: the online-failure-free (C1), online-recovery (C2) the recovery (C3). Based on this classification, Pandora borrows C1 from FORD; Pandora also borrows C2 from FORD but significantly reworks it, making it efficiently recoverable. One of the other limitations of FORD is that it lacks a recovery component (C3). In Pandora, we introduce a recovery algorithm that works over one-sided RDMA.

4.3.1 Making FORD Efficiently Recoverable

In Section 4.1, we asserted that the FORD protocol prohibits fast recovery on a compute failure because it first latches objects and later it writes logs for said latches. We elaborate on this issue and present our solution *Implicit latch logging*.

4.3.1.1 Problem: Stray latches

On a compute failure, it is possible that several objects are latched but there is no log that points to these latches. We call these *stray latches*. First, we discuss why stray latches prohibit fast recovery. Then we delve more into the problem, arguing that it is more subtle than it looks, with its roots stemming from adopting disaggregated memory.

Impact on Recovery. Stray latches create two related problems. First, we cannot simply unlatch while other compute servers are executing transactions, as we cannot differentiate between the stray latches and the regular latches of the live servers. Second, we need to scan the entire memory to find the stray latches, which can take seconds: e.g., scanning 100 GiBs through a 100Gbps network requires at least 8 seconds. Hence, we must block the entire system for several seconds.

Understanding the problem. To modify an object, FORD issues an RDMA CAS to latch it first, and then an RDMA Read to read it. Because RDMA guarantees that the two messages will be delivered in order, we are certain that we read the object only after latching it. Furthermore, only after the RDMA Read has returned can we perform the undo logging. This is because undo logs store the previous value (so that they can "undo" the change). Note that, had we read the value without first latching it, we would not be able to log it, because it would be possible to log a different value than the one we latched. Thus, we need to latch before reading, and read before logging. This is why there are stray latches.

The role of RDMA. This ordering conundrum does not exist in the traditional non-disaggregated architecture, because an RPC can execute all three tasks – latching, reading, and logging – in the same step. Crucially this step is atomic with respect to failures. I.e., if the server that executes the RPC fails, then neither the log nor the latch will be visible. In contrast, with one-sided RDMA-access to a remote memory server, we do not have the luxury of performing these multi-step functions in a failure atomic manner. We expect this to become a recurring problem as more and more applications are ported to disaggregated memory.

Summary. Stray latches are a non-trivial problem that occurs because, in a disaggregated architecture, it is not possible to perform multi-step functions in a failure atomic manner. Crucially, this problem prohibits fast recovery.

4.3.1.2 Our solution: Implicit latch logging

To solve this problem, we assign a unique 16-bit process-id to each compute server and mandate that latches include the process-id of their owner compute server. On a compute server failure, we need not scan the entire memory in a blocking manner to release its stray latches. Instead, we enable other transactions to *steal* these latches. We call this technique *Implicit latch logging* because we have repurposed the process-id (added to the latch) to signify whether or not the latch is stale, avoiding the need for

explicit logging.

How does stealing work? Recall that the coordinator issues an RDMA CAS to latch an object. When the RDMA CAS fails, it returns the value of the latch, which includes its owner process-id. We check this process-id against a series of the *failed-ids*, i.e., an array that contains the process-ids of all previously failed compute servers. If we discover that the latch is stray, we execute one more RDMA CAS to steal it. Notably, stray latches can also cause Reads to abort, during both execution and validation phase. To avoid this, we again check the failed-ids, and if the latch is found to be stray, we proceed as if the object were not latched at all.

Overhead. The overhead of this approach is: 1) a check against the failed-ids, incurred only when accessing a latched object and 2) an extra RDMA CAS when finding a stray latch. Note that actually finding a stray latch is extremely rare, because we execute millions of transactions per second, while we may only get one failure every a few hours (depending on the number of compute servers [26]).

Recycling process-ids. We must ensure that a failed process-id cannot be assigned to another compute server, until all of its stray latches are unlatched. We ensure this as follows. We use 16 bits to represent process-ids, allowing for 64Ki compute servers to join over the lifetime of the system. Although we expect 64Ki process-ids to be plenty, they might outlast the utility of a long-running system. As such, we implemented a background mechanism that scans the memory and unlatches all stray latches, allowing to recycle failed process-ids. We trigger this mechanism if more than 95% of available process-ids are used.

Notably, as more compute servers fail over time, we must ensure that the overhead of checking the failed-ids stays constant. We achieve this by implementing failed-ids as a compact bitset with 64Ki entries.

Updating failed-ids. After a compute server failure, the recovery protocol is responsible for notifying all alive compute servers, so that they can update their failed-ids. We discuss this further in the next section.

Summary. We presented implicit latch logging, a technique that associates each latch with the unique process-id of its owner compute server. This allows us to detect which latches are stray and steal them. We use a large enough number for process-ids to ensure that we will not need to recycle them, but have a contingency plan for that. Crucially, implicit latch logging enables recovering from a compute failure without interrupting the rest of the system.

4.3.1.3 Problem: Logging aborted transactions.

In FORD it is possible for a logged transaction to be aborted. The problem is that at recovery-time it is impossible to differentiate between committed and aborted logged transactions. This prevents correct recovery. For instance, consider a failed compute server C, which has logged a write to object X in one of its transactions. Also, assume that during recovery, we see that X has been modified. It is impossible for us to know whether X has been modified by C or not. This is because, it is possible that C's transaction aborted, unlatching X and then a different compute server latched and updated X. As we will see, when discussing our recovery protocol (§4.3.2.2), recovery hinges on knowing whether C is the one that modified X, so that we can undo the modification, if needed.

Notably, we realized this was an issue after our validation revealed three bugs that are caused by this problem. We discuss further in Section 5.3.

4.3.1.4 Solution: Logging phase.

Firstly, note that we cannot solve this problem by relying on the fact that latches include the process-id of their owner, because we do not latch backups. Therefore, if the memory server that serves as the primary for object X fails, we will still face the same problem.

There are multiple ways to solve this problem. We add an extra *logging* phase in-between validation and commit/abort. This phase is executed only if validation succeeds. To minimize the performance overhead of this extra phase, we change FORD's logging scheme such that the overall logging overhead is reduced.

FORD writes a log in each replica of each object in its write-set. For example, assume a transaction that writes X and Y with a replication degree of 3 (i.e., $f+1=3$), where X is replicated in memory servers 0,1,2 and Y is replicated in memory servers 3,4,5. FORD will log X in 0,1,2 and Y in 3,4,5. We take a different approach. For each compute server, we specify f+1 memory servers that hold its logs. Therefore, in the above example, both writes to X and Y will be logged in the same three servers. This is a well-established technique [195].

As we log after validation, at which point we know the entire write-set, we can log all writes with the same RDMA Write amortizing its overheads. Therefore, the total cost of logging in our technique is always $f+1$ RDMA Writes as opposed to FORD's $f+1$ RDMA Writes per object in the write-set. This technique also makes the recovery

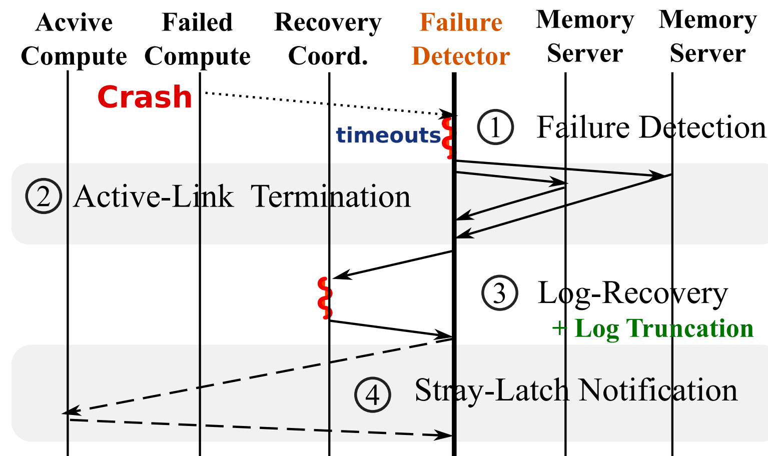


Figure 4.3: Recovery Protocol

protocol slightly simpler, as all the logs of a compute server are gathered in the same $f+1$ memory servers.

4.3.2 Recovery Protocol

In the previous section, we ensured that fast and non-blocking recovery is possible. In this section, we guarantee that it is also correct. We start by specifying four correctness criteria. Then we describe non-blocking recovery from compute failures. Finally, we provide a brief description of how we handle memory failures.

4.3.2.1 Correctness criteria

Before we state the correctness criteria, we must first introduce some definitions. A failed compute server, C, may have been working on a number of transactions, before failing. We refer to these as *stray transactions (Stray-Txs)*. There are three side-effects of Stray-Txs that must be addressed: 1) stray latches on objects 2) updates on objects (during commit phase) and 3) communication with the client to notify it of commit or abort. We differentiate between two types of Stray-Txs, the *Logged-Stray-Txs* for which C has written a log, and the *NotLogged-Stray-Txs*, for which C had not yet reached the point of writing a log. The dichotomy is important, as transactions that have written a log can have all three of the side-effects, while the NotLogged-Stray-Txs can only have stray latches.

Note four correctness criteria for the recovery algorithm, after the failure of compute server C.

(Cor1) Before trying to recover the Stray-Txs of C, we must ensure that C cannot access memory anymore. This ensures that memory will not be compromised by unreliable failure detection.

(Cor2) We must either roll back or forward all Logged-Stray-Txs, to ensure that either all or none of their updates are applied to objects.

(Cor3) We must not roll back a Logged-Stray-Tx, if it has notified its client that it has committed. And vice versa, we cannot roll forward, if the client has been notified of an abort.

(Cor4) We can only steal the stray latches from NotLogged-Stray-Txs. This is because Logged-Stray-Txs may have also updated some of the latched objects and thus, stealing could leave the memory in an inconsistent state.

4.3.2.2 Recovering from compute failures

Figure 4.3 illustrates the protocol, which comprises four steps. Below we provide an overview of each step.

(1) Failure Detection. The first step of the protocol is initiated by a fault detector (FD), which detects a crash on a compute server. Our protocol can work with any off-the-shelf FD. For our evaluation, we have implemented a heartbeat-based FD, which exchanges heartbeats with compute servers and reports failure after a time-out of 5ms.

(2) Active-Link Termination. Notably, any FD can have false positives, i.e., it can mistakenly deem compute server C as failed. Recall correctness criterion (Cor1): before recovering the Stray-Txs of C, we must ensure that C can no longer access memory. In a non-disaggregated system, this is typically achieved by rejecting RPCs from servers that are not in the stable configuration [122]. To achieve the same effect, we revoke C's RDMA rights, ensuring that any future requests from C will get dropped. We call this *active-link termination*. Recall that memory servers have some low-power, cheap compute for network management. We implement active-link termination by sending a link-termination RPC to this compute.

(3) Log Recovery. Again, assume that compute server C has failed. For each of failed C's Logged-Stray-Txs we will make a decision, either roll it forward or backward, satisfying criterion (Cor2). Recall that in its commit phase, the transaction will update all replicas of an object (i.e., it applies its writes). To also address criterion (Cor3), we make the following two assertions on the protocol 1) if all replicas of all objects in the

write-set are updated, then it is possible that the client has received a commit-ack. 2) If any of the objects in the write-set are updated, then it is impossible that the client has received an abort-ack. Based on these, we can assert that we can safely roll-forward all Logged-Stray-Txs that have updated all objects in their write-set in all replicas because the commit-ack is possible, but the abort-ack is impossible. We roll-back all other Logged-Stray-Txs. This is correct because it is impossible that we have sent a commit-ack for these transactions.

In practice, we implement log recovery as follows. First, we spawn a thread which we call Recovery Coordinator (RC). Recall that for any compute server, we write its transaction logs in $f+1$ specific memory servers (§ 4.3.1.4). Therefore, the RC can read all logs by issuing $f+1$ RDMA Reads. Using the logs, the RC recreates the write-set of each Logged-Stray-Tx. Then for each Logged-Stray-Tx it issues an RDMA Read for every replica of every object in its write-set, so that it can check if it has been updated. Specifically, each object has a version, so we simply read the version and compare it with the version in the undo logs. Then, for each transaction that has updated all replicas of all writes in its write-set, we simply unlatch its latches with an RDMA Write to each replica. For the rest of the transaction, we also unlatch all objects, but also use the undo log to roll back any updated objects.

Note here the importance of the second fix in FORD's online-recovery. Had we not done the logging after validation, it would be impossible to differentiate between an object that is updated by a Stray-Tx or by a live transaction from an alive compute server. This is because in FORD it is possible to log an object, and then later abort and unlatch it. However, the log would remain.

(4) Stray latch notification. Finally, we notify all compute servers of a failure so that they can start stealing the stray latches of the failed server. Recall the first correctness criterion: we can only steal stray latches of not-logged stray transactions. For this reason, it is crucial that we only perform the stray latch notification after log recovery.

4.3.2.3 Idempotent Recovery

Pandora ensures idempotent recovery, enabling any step of the end-to-end recovery algorithm to be re-executed. This capability is essential to tolerating failures during the recovery phase, given that the recovery coordinator operates within a standard compute server. For instance, in cases where compute failures can cause log recovery to stall, necessitating re-execution, Pandora allows for the re-execution of the log-recovery

step until the final acknowledgment is received from the recovery coordinator. To guarantee idempotent correctness, Pandora truncates all logs from the failed transaction coordinators before sending the Stray-Latch notifications (refer to Figure 4.3).

4.3.2.4 Failure Detector Availability

The availability of the failure detector is crucial for the end-to-end recovery algorithm. Pandora ensures detector availability by replicating it on ZooKeeper quorums [110]. To achieve this, we decouple the detector's program state and migrate it to ZooKeeper replicas. This approach not only tolerates detector failures but also eliminates false negatives resulting from compute server and network delays. With quorum replication, a compute node is considered failed only if it fails to reach a majority of the replicas (it is considered alive as long as it reaches a majority of the quorums).

The trade-off in this approach is that compute nodes now send heartbeat messages to all ZooKeeper replicas or a majority in the case of failures, introducing some latency into the end-to-end recovery algorithms. However, this latency is not expected to significantly impact recovery time and can be eliminated by alternative solutions [95].

4.3.2.5 Recovering from memory failures

This work focuses on compute failures because they present the opportunity for non-blocking recovery. We handle a memory server failure in three steps. First, we detect the failure and notify all compute servers. For each object whose primary is lost, compute servers know deterministically which is the new primary using consistent hashing [121]. Notably, we do not need to recover any transactions, because all compute servers are still alive, and have full knowledge of the state of their transactions. After learning of the memory failure, each compute server makes a decision for each of its transactions, using the same criterion as log recovery: it commits transactions that have updated all live replicas and aborts the rest. Once all compute servers complete this step, operation can resume. In the case where memory and compute servers fail together, we execute both protocols independently. Notably, we do not handle adding memory servers.

4.4 Evaluating Pandora: Goals and Methodology

Recall that the goal of this work is to achieve fast and correct recovery for transactions on DKVSes. Therefore, we conduct experiments to answer four key questions.

Validation. Recovering transactions correctly on DKVSes involves subtlety. Is Pandora actually recoverable? We validate Pandora as well as the state-of-the-art protocol FORD [219] using our litmus-test-based validation framework. We will extensively discuss validation in Chapter 5.

Recovery latency. Reducing recovery latency is one of the key goals of our work. What do we mean by recovery latency precisely? Recall that our proposed techniques do not stop the entire KVS on a failure, but transactions whose coordinators fail are affected. The recovery latency refers to the delay seen by such transactions that are affected by failures. We show the recovery latency of Pandora.

Fail-over throughput. When a failure does happen, can our techniques ensure minimal disruption? We show the fail-over throughout – the throughput of Pandora when it is recovering from a failure.

Steady-state throughput. How much overhead does Pandora impose on steady-state failure-free execution compared to the existing state-of-the art? We show the steady-state throughput of Pandora and compare it with the FORD baseline.

Before diving into our experimental evaluation we first explain our experimental setup, workloads and methodology.

4.4.1 Methodology

Setup. We conducted our experiments on a cluster of 5 servers in CloudLab [62]. Each server is an r650 node in the Clemson cluster. A server can play the role of either a compute or a memory server. The configuration is different in different experiments. We use a dedicated server for our failure detector and recovery manager. We will explain the different configurations separately in each experiment. Each machine in our setup runs Ubuntu 18.04 and is equipped with two 36-core Intel Platinum CPUs with two hardware threads per core. Furthermore, each machine has 256GB of system memory and a dual-port 100Gbps Mellanox ConnectX-6 Infiniband NIC.

Protocols: Baseline vs Pandora. For our evaluation, we have adopted the in-memory version of FORD KVS [219] as the system for deploying the protocols. (Recall that FORD is the only fully one-sided transactional DKVS in the literature.) Because FORD misses the recovery part of the protocol, we integrated our recovery algorithm to FORD to make it our *Baseline*. We compare this Baseline protocol against Pandora, which takes the online-failure-free component from FORD, but significantly improves upon

Bench\Coordinators	1	8	64	128	256	512
TPC-C	8 us	22 us	158 us	272 us	563 us	4951 us
SmallBank	8 us	139 us	232 us	424 us	876 us	5272 us
TATP	9 us	20 us	131 us	513 us	1039 us	2236 us
MicroBench	10 us	21 us	119 us	474 us	1001 us	2043 us

Table 4.1: Recovery latency (in microseconds) while increasing the number of outstanding coordinators per compute node.

the online-recovery component of FORD to speed up recovery via implicit latch logging, and introduces a new recovery component.

Workloads. For validating the Baseline and Pandora, we used our litmus tests, which we will describe in the next section. For performance evaluation, we use the same three standard OLTP benchmarks that were used by FORD: TPC-C [4], TATP [2], and SmallBank [3]. These benchmarks have 8B keys. The values are 671B, 48B, and 16B, respectively. In addition to these standard benchmarks, we have used a microbenchmark with 8B keys and 48B values in which read/write ratios can be adjusted. It is worth noting that each workload runs a different number of transaction coordinators (which we explicitly specify). Each coordinator is a separate C++ coroutine, and we run up to 7 of them concurrently within a single thread. Unless mentioned otherwise, each of our workloads runs on 16 hardware threads.

Workloads Parameters. TATP, SmallBank, and TPC-C consist of 4, 2, and 9 tables, respectively. In TATP, 80% of the transactions are read-only. In contrast, both SmallBank and TPC-C have high write ratios – 85% and 95%, respectively.

4.5 Experimental Evaluation

The goal of our experimental evaluation is to compare the Baseline against Pandora on the following performance metrics: recovery latency, fail-over throughput, and steady-state throughput.

4.5.1 Recovery Latency

In this section, we report recovery latency for Baseline and Pandora. Recall that FORD's design incurs a significant overhead on recovery. On a failure, the entire KVS must be stopped and searched to detect stray latches. We observe that these overheads are

in the order of seconds. Specifically, in our measurements, a recovery program that runs on a single thread while searching over the KVS using one-sided reads adds up to around 5 seconds for 1 million keys. While more threads could be used, the latency grows linearly with the number of keys. Overall, this is impractical for today's KVSes that demand high availability, hence we do not explore the Baseline's recovery latency any further. Instead, we focus on the recovery latency for Pandora. But before this, we briefly describe how we emulate failures.

Emulating Failures. In this experiment, we emulate a failure by stopping a process at a selected point in time, which implicitly stops all the in-flight transactions running within that process. The failure detector (FD) identifies these failures using timeouts and requests the recovery coordinator to perform recovery for each of the failed coordinators. We use *5ms* timeouts in the FD.

Pandora. Recall that Pandora introduces implicit latch logging, a fast recovery technique that moves the recovery for stray latches out of the critical path of failures. Table 4.1 shows the recovery latency for each benchmark with respect to different numbers of outstanding transactions (i.e., transaction coordinators). Specifically, in TPC-C, SmallBank, and TATP recovery takes 5ms, 5.3ms, and 2.2ms respectively (with 512 outstanding transactions). In addition, our micro-benchmark with 100% writes shows 2ms latency. The latency represents the time spent in the log recovery step of the recovery protocol.

As we can see from the table, the recovery latency is within milliseconds, which is three orders of magnitude smaller than the recovery latency in Baseline (FORD). Recall that the recovery latency for the Baseline is in the order of seconds as it needs to scan the entire KVS for latch recovery while stopping the remaining transactions. As expected, we observe that recovery latency increases with the number of transaction coordinators, as more outstanding transactions must be recovered upon a compute server.

Naive Logging Scheme. In addition to our main techniques, we also evaluate a naive scheme that adds extra logging to the protocol should latches be recovered in the recovery phase. Recall that the undo logging scheme used in FORD is not sufficient for recovering latches. Therefore this naive scheme adds extra logging before the latch operation executes on the transaction coordinator. We extend our recovery protocol with this extra logging to recover latches. With the highest number of outstanding transactions (512), recovery latency for the TATP, TPC-C, SmallBank, and MicroBench reaches 10ms, 13ms, 2.7ms, and 2.5ms respectively.

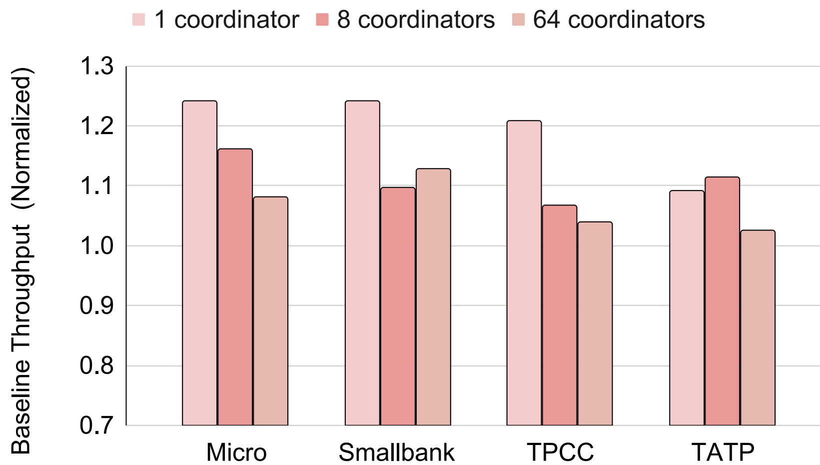


Figure 4.4: Baseline throughput normalized to naive logging (aka Log-A-Latch)

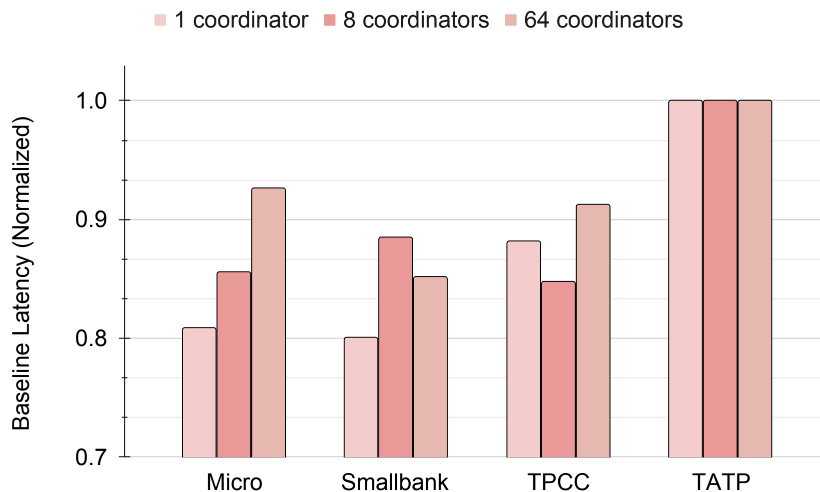


Figure 4.5: Baseline latency normalized to naive logging scheme (aka Log-A-Latch)

Summary. Overall, these results show that implicit latch logging significantly reduces recovery latency, substantiating our argument that, we can recover fast from a compute failure on DKVSes. Next, we will look into how much overhead these protocols impose on failure-free stable-state execution.

4.5.2 Steady-State Throughput

Pandora's implicit latch logging (ILL) offloads the recovery of (stray) latches to the execution phase of the transaction, potentially adding extra overhead on steady-state protocol. Recall that ILL adds three extra steps to the steady-state protocol: (1) latching with process-ids, (2) a check against the failed-ids, and (3) releasing stray latches.

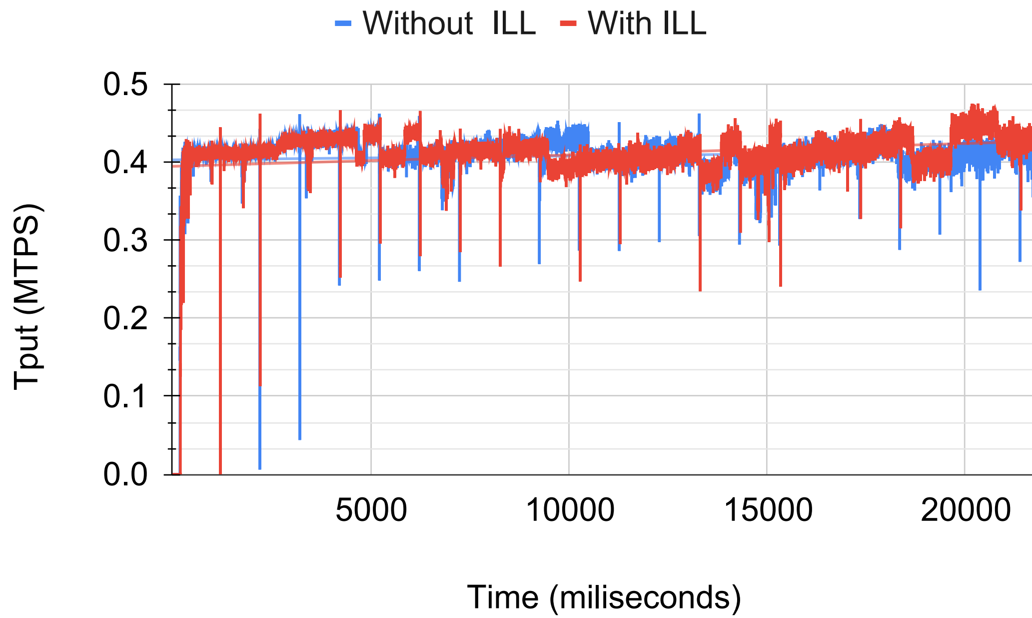


Figure 4.6: Steady-state overhead of Pandora

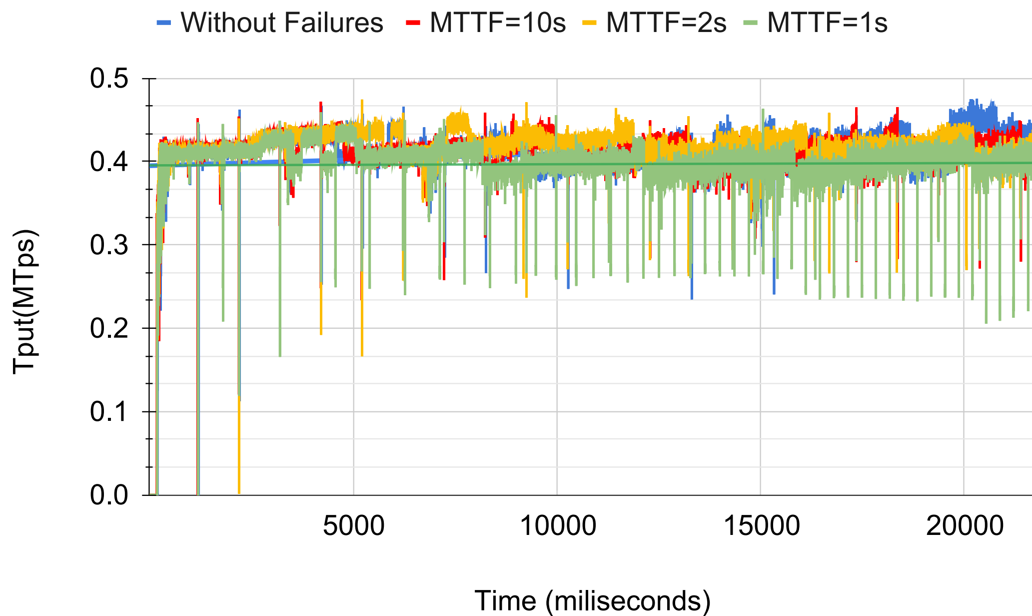


Figure 4.7: Steady-state overhead of Pandora

Notably, the overhead of the last operation is only visible when there are actual failures.

First, we evaluate the steady-state overhead of ILL (only (1) and (2)). Second, we measure the overhead of ILL under failures.

ILL under no failures. For this experiment, we use our micro-benchmark with 16

transaction coordinators. Figure 4.6 shows the throughput over time without (blue) and with (red) ILL. Note that the throughput difference is very negligible. This is because the failed-id list is empty hence we do not incur any extra round trip overhead for stealing latches. It is worth noting that each failed-id bitfield lookup (with $\mathscr{O}(1)$ complexity) only adds a few nanoseconds on every failed latch (and reads), which is insignificant compared to the round trip latencies that are in the order of microseconds.

ILL under failures. In this experiment, we measure the end-to-end steady-state overhead of ILL under failures. Recall that after failures stealing the latch adds an extra round trip. To measure the overhead, we ran the same experiment with failures that stop (then recover) half of the coordinators in the setup. We then reduce the Mean Time To Failure (MTTF) and rerun the experiment. Lower MTTF means that the number of stray latches in the DKVS is higher and the time to recover these latches before the next failure is lower.

Figure 4.7 shows the transaction throughput without failures (blue), with *MTTF=10s* (red), *MTTF=2s* (yellow), and *MTTF=1s* (green). It is worth noting that the typical MTTF in the datacenter is in the range of minutes [26], and MTTF $<$ 10s is highly unlikely. As we can see, it is clear that ILL adds insignificant overhead under failures. This is because only just a few stray latches must actually be recovered and that overhead is uniformly distributed over the run of the experiment.

Naive Logging. In addition to the proposed technique, we show the steady-state overhead of the naive scheme that we discussed. Recall that for this scheme to work we need an additional logging round trip for each latch in the steady-state execution phase. Hence, the steady-state throughout should be lower than that of the baseline FORD's throughput. Figure 4.4 shows the steady-state throughput normalized to FORD. Logging adds a 25% throughput overhead on the standard benchmarks. Smallbank, TPC-C, and TATP incur average throughput overheads of 24%, 20%, and 9% respectively. Microbenchmarks with 100% writes incur an overhead of 24%.

We observe that logging overhead generally increases with increasing write ratios. For instance, TATP, which is mostly read-only, shows lesser overhead than write-intensive workloads like SmallBank. On the other hand, some write-intensive workloads like TPC-C show lesser overhead than anticipated because of one-sided read overhead that is not proportional to the actual read/write ratio present in the benchmarks.

Summary. ILL adds minimal overhead over the FORD baseline while significantly reducing the recovery latency. Next, we look at the fail-over throughput of our technique.

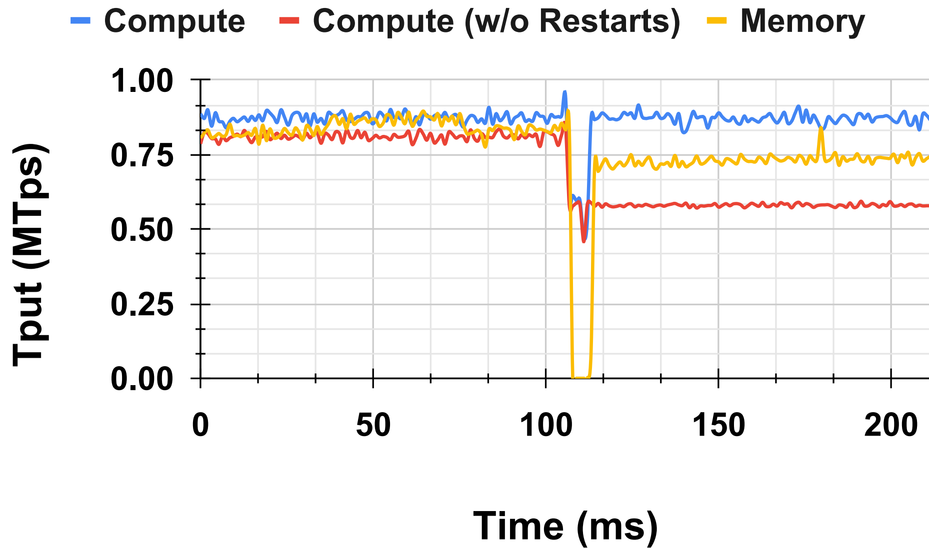


Figure 4.8: Micro fail-over throughput

4.5.3 Fail-Over Throughput

The fail-over throughput is the difference between the throughput while recovering from a failure and the fault-free steady-state throughput. To measure this for Pandora, we conducted an end-to-end experiment to show the impact of our fast recovery that does not stop the entire KVS.

For this experiment, we set up a cluster of five machines with two memory nodes and two compute nodes; the fifth server is used to run the failure detector. We use our standard benchmarks while running 64 transaction coordinators on each machine. We emulate a failure by crashing one compute node while measuring the throughput of the rest of the KVS. Recall that the failure detector identifies the failure after waiting for the timeout (i.e., 5ms in our case) and initiates the recovery coordinator. For this experiment, we use the same failed machine to run the recovery coordinator.

Unlike blocking (i.e., ”stop-the-world”) type recoveries as in the Baseline (or traditional monolithic server deployments [192, 68, 59, 60]), our recovery need not stop the entire KVS for compute failures. Indeed, our microbenchmark in Figure 4.8 (blue line) shows that the throughput of Pandora does not drop to zero but rather drops to about two-thirds of the original throughput after the emulated crash. Similarly, Figure 4.9, Figure 4.10, and Figure 4.11 respectively show the fail-over throughput of Smallbank, TATP, and TPCC. Recall that Pandora can handle memory failures as all-compute failure that requires stopping the entire KVS to update the new replica configuration. Thus, in our benchmarks (yellow line) fail-over throughput drops to zero.

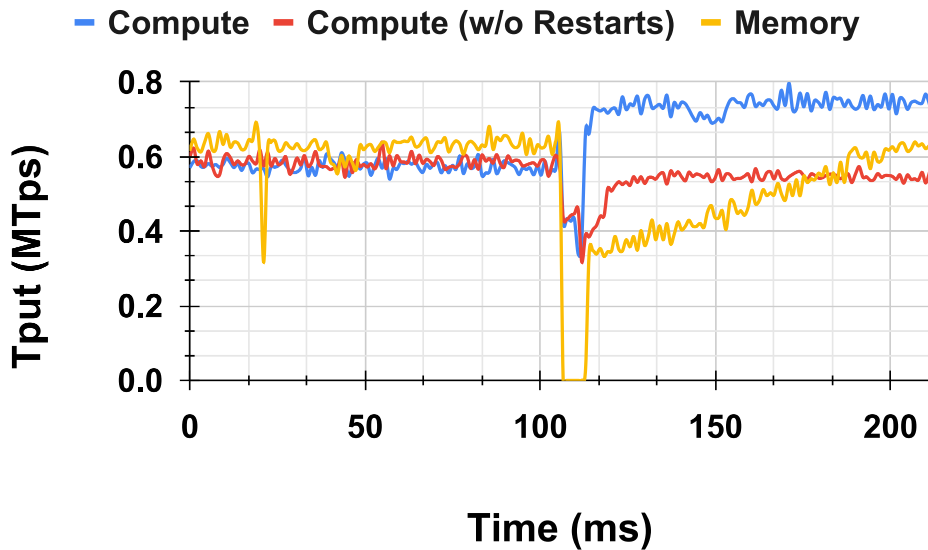


Figure 4.9: SmallBank fail-over tput

Post-Failure throughput. Recovery impacts not only the fail-over transactional throughput but also the post-failure throughput. If the recovery cannot restore the lost compute resources, the post-failure throughput is likely to drop in proportion to the percentage of lost coordinators. In some cases, the post-failure throughput hinges on the ability to restore the failed coordinators after the recovery process. In such a scheme, the KVS can either use the freed-up resources from failed coordinators or standby backup resources. Reusing resources from failed coordinators is possible for software crashes. Figure 4.8 shows two scenarios: the blue line represents the situation when there is a fault followed by recovery, but the failed resource is not reused. The red line denotes the case in which the failed resources are reused, and hence the post-recovery throughput matches the pre-failure throughput. It is worth noting that the failed coordinators are brought back in less than 10ms after the failure.

Moreover, post-failure throughput is impacted by system bandwidth limitations. For example, in scenarios with high load where transactions operate significantly below optimal throughput, due to a multitude of coordinators competing for memory bandwidth, a loss of a compute node might paradoxically elevate the application's throughput. This phenomenon manifests in certain workloads, resulting in post-failure throughput exceeding the steady-state throughput observed pre-failure. However, this elevation is primarily attributed to bandwidth constraints and can be mitigated by reducing the number of coordinators active on the compute nodes. For instance, Figure 4.12 depicts the Smallbank benchmark with half the number of coordinators. In this configuration, Pandora effectively restores the post-failure throughput to its pre-failure levels.

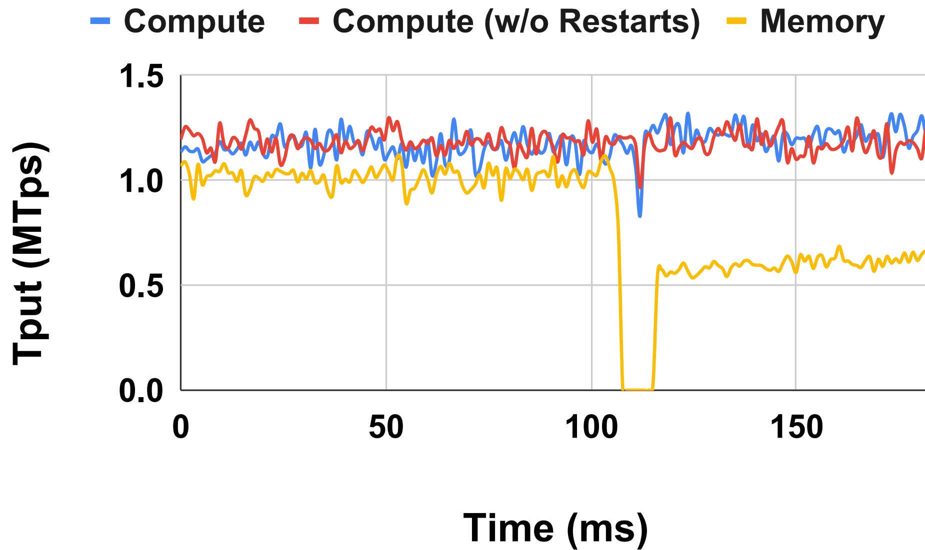


Figure 4.10: TATP fail-over throughput

Distributed Failure Detector. Recall that replicating FD using zookeeper quorums affects recovery time (Section 4.3.2.4). However, even when using three replicas, Pandora consistently recovers in under 20ms.

Sensitivity to stalls. Let us consider the situation in which a transaction T1 latches an object X during transaction execution, and then, is forced to abort due to a failure. Recall that the failure would trigger a recovery operation (Section 4.3.2). But before the recovery can complete, suppose another transaction T2 accesses the same object X during transaction execution. At this point there are two options: abort transaction T2 or stall T2 until recovery is complete. Thus far, we have assumed the former, letting other transactions that do not conflict with object X to execute and commit.

In this experiment, we explore the stalling approach: we assume that a transaction that needs to access an object that needs to be recovered is delayed until recovery completes. Naturally, using this approach impacts fail-over throughput, and its impact is proportional to the recovery latency. To show the sensitivity of fail-over throughput to different recovery latencies, we ran two experiments with a microbenchmark with 100% writes on the same setup as in the previous experiment. To emulate failures, we crash half of the coordinators at random times. In our first experiment, we used 1000 hot objects/keys (Figure 4.13). As we can see, without fast recovery provided by our mechanisms, throughput drops to zero: the high recovery latency and the high conflict rate eventually block all of the transactions. In contrast, with fast recovery, there is initially a steep drop in throughput while the recovery is taking place – but as soon

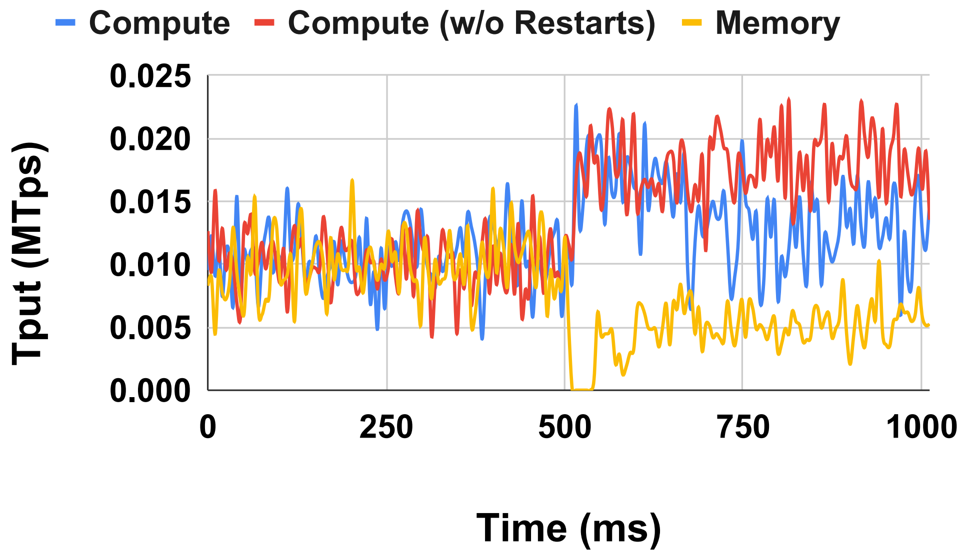


Figure 4.11: TPCC fail-over throughput

as recovery completes, the throughput increases and stabilizes. We also conducted a similar experiment with 100000 hot objects to reduce the number of transaction conflicts (Figure 4.14). The reduced probability of conflicts means that, even with slow recovery, there are non-conflicting transactions to execute. Thus, the throughput, instead of dropping to zero immediately, drops gradually. With fast recovery, however, there is no drop, while the fail-over throughout depends only on the number of failed coordinators, which can be restored later.

4.6 Summary

We have observed that disaggregated memory presents an opportunity to handle compute failures efficiently while ensuring high availability. To seize this opportunity, we have proposed Pandora, the first one-sided transactional protocol that ensures correct, non-blocking, and fast recovery in DKVSes. Pandora makes two key innovations: Implicit Latch Logging and a novel one-sided recovery algorithm. For correctness, we introduced an end-to-end litmus testing framework that revealed multiple bugs in the FORD protocol, which we have addressed in Pandora. Our experimental evaluation demonstrates that on a compute server failure, Pandora does not unnecessarily block transactions on alive servers, and recovers fast (in just a few milliseconds). Crucially, we have also shown that the recovery enhancements of Pandora incur minimal overhead on failure-free execution.

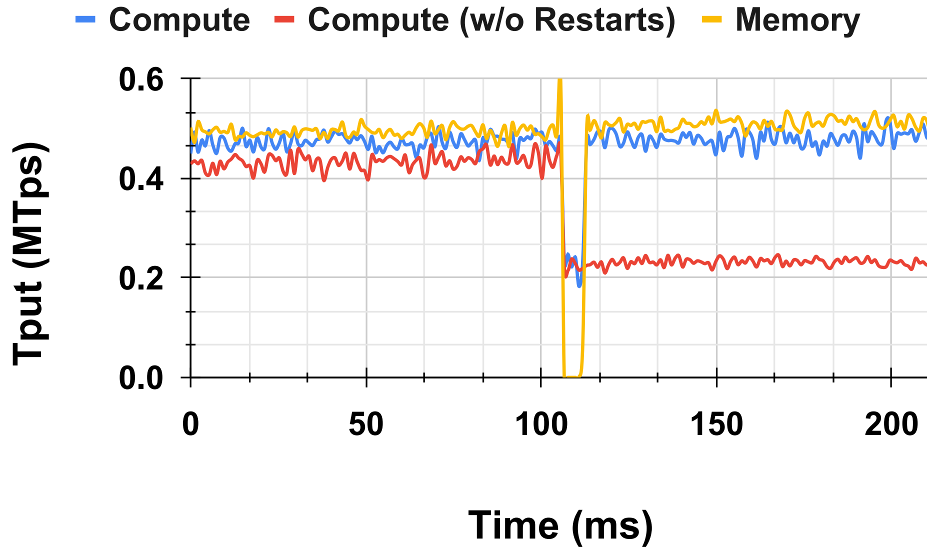


Figure 4.12: Smallbank fail-over throughput with low contention

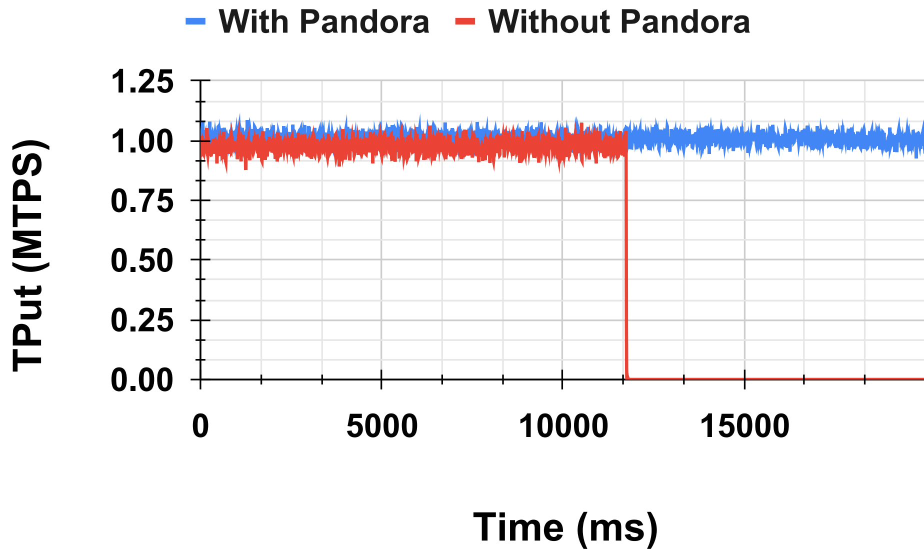


Figure 4.13: microbenchmark with stalls or hot objects=1000

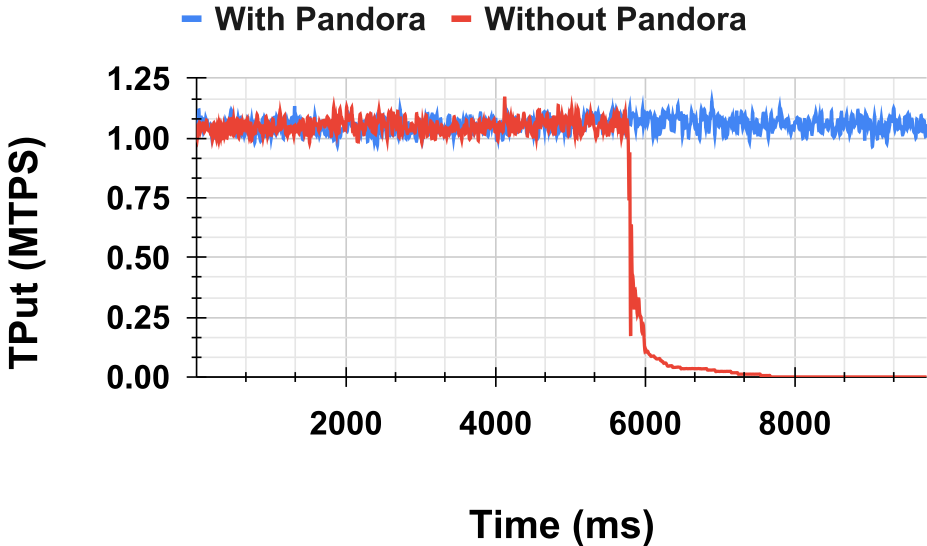


Figure 4.14: microbenchmark with stalls or hot objects=100000

Chapter 5

DART: Validating Transactional Databases with Target Application-Centric Recurrent Testing

5.1 Introduction

Verifying transactional protocols is notoriously hard. Although there is a rich literature on formally verifying *models* of transactional protocols using manual and automated techniques [32, 134], our focus is on validating real *implementations* of the protocols. End-to-end testing with randomly injected failures has proven to be very effective in revealing bugs in transactional protocols used in databases [115, 152, 127, 28].

In this chapter, we discuss, DART, a target (black-box) litmus-testing framework specifically designed for validating end-to-end correctness of transaction protocols. DART's end-to-end testing consists of two steps: offline test generation and online execution. In the offline path, DART uses the consistency model to generate for each test a set of litmus transactions and a corresponding assert transaction(s). In the online path, DART (recurrently) runs the litmus transactions and the assert transaction(s) on many application instances with randomly injected failures into the database; Any violation of the consistency model is revealed in the read state of the assert transaction(s).

DART offers two benefits: first, DART scales well since the validation hinges solely on the read state of the assert transactions, and second, DART incurs zero integration cost as it requires no changes to the application programming interface (API).

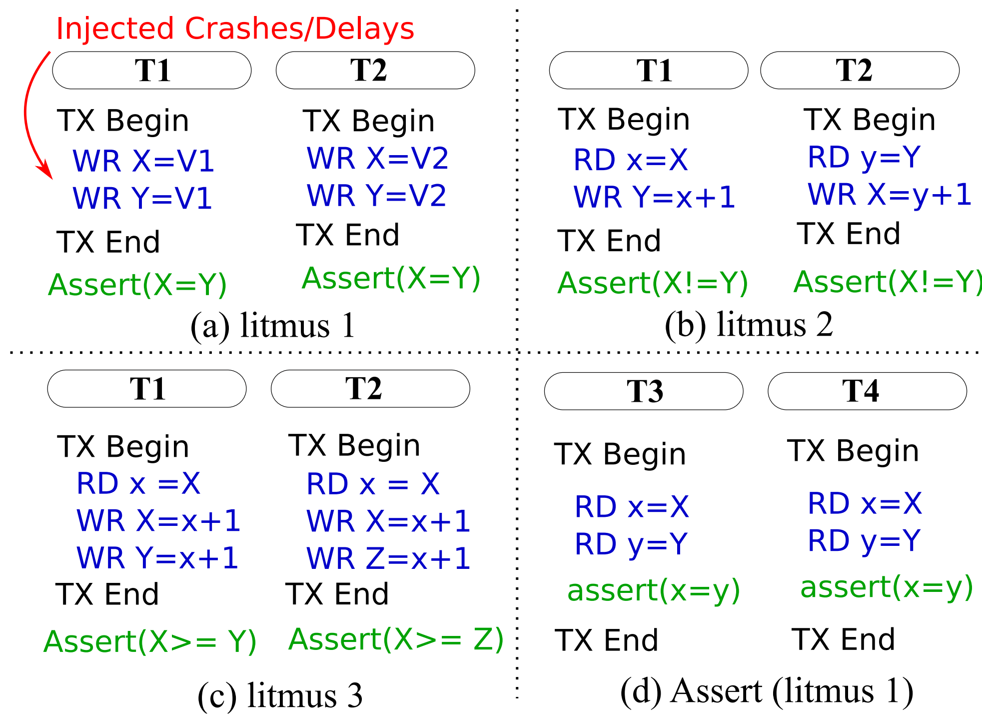


Figure 5.1: Basic litmus tests with application-observable states.

5.2 Method

The most commonly used technique for testing databases is Adya's Histories [8, 127]. The idea is to run a number of randomly generated transactions, collect a rich trace of data and metadata for each run (the history), and use the history to determine (violations in) the consistency model enforced by the database. However, existing frameworks – because they need to collect histories – tend to be heavyweight, hard to integrate and scale.

There is an alternate method to validating databases – one based on *application-observable state* [53] rather than histories. The idea is to carefully construct transactions so that the values of the objects reveal the consistency model, and consequently reveal bugs in the protocol (if it does not match the intended consistency model). Crooks et al. [53] showed theoretically how the application-state-based approach can be equally effective as the histories-based approach while being significantly more lightweight and less costly. (It is also worth noting an analogous approach of *litmus testing* [13, 146, 58] has been extremely effective for validating shared-memory consistency models.) However, being a conceptual framework, it does not contain a suite of tests or a tool that can be readily used for testing protocols.

To the best of our knowledge, this is the first work to create transactional (litmus)

tests for black-box testing of real-world protocols. Figure 5.1 lists 3 basic litmus tests that we have developed in our framework for validating transaction protocols with (strict) serializability, the intended consistency model. These litmus tests are derived from the formal definition described in Section 2.3. They comprehensively cover all possible scenarios of cyclic violations (cycles) in the serialization graph with a minimal number of dependency edges. Our basic litmus tests target three types of cyclic scenarios: *direct-write*, *read-write*, and *indirect-write* cycles, addressing dependency violations.

Limitations and Challenges. Our tests are sound and complete in identifying serialization bugs within an arbitrary testing time. However, these straightforward tests may not always be exhaustive in detecting real-time ordering bugs. It's important to note that, while not universally necessary, real-time ordering is crucial for testing strictly serializable transactions. Therefore, before running out tests at scale, we initially ran our tests in a single-threaded setup to carefully analyze whether there were any real-time ordering violations in the transaction protocols. Specifically, we use traces and the physical timestamp of each transaction to verify if any read has not captured the immediately written values.

5.2.1 Application-Centric Assertions

In addition to the litmus tests, we also create their matching application-centric assertions. The creation of an assertion for each test is a non-trivial task and is highly contingent upon the specific test. In this thesis, we opt for the minimal assertable version for each test, which represents the litmus test with the fewest operations.

We use special read-only *transactions* for realizing the assertions. For example, the first litmus (Figure 5.1(a)) assigns a value of V1 to both variables X and Y in the first transaction and assigns of value of V2 to both variables in the second transaction. Strict serializability mandates that the values of X and Y should be equal at the end of each transaction. This is exactly what we assert with our read-only transaction (Figure 5.1(d)). To test the steady-state protocol together with the recovery protocol, we randomly inject crashes after every statement in a transaction.

While litmus transactions run regularly on numerous instances, assert transactions are only required to run at selective or random intervals. However, running assert transactions at regular intervals can minimize testing time.

Litmus	Bugs (Category-Source)	Description	Fix(es)
Litmus-1 (Direct-Write Cycles)	Complicit Aborts (C1 - Baseline/Pandora)	Releasing unset latches in the abort path	Unlatch only the acquired latches during execution in the abort path
	Missing Actions (C2 - Baseline)	Omitting logging of inserts	Add inserts into undo logs.
Litmus-2 (Read-Write Cycles)	Covert Latches (C1- Baseline/Pandora)	Not checking the latch value in the validation phase	Read and check latches of read-only data during the validation phase
	Relaxed Latches (C1 - Baseline/Pandora)	Relaxing the order of latches and validation in the commit path	Grab all latches before validation
Litmus-3 (Indirect-Write Cycles)	Lost Decision (C2 - Baseline/Pandora)	Logs for a transactions that aborted without being able to tell if it hs updated its objects	Add log phase if validation succeeds (Section 4.3.1.4)
	Logging without latching (C2 - Baseline/Pandora)	Loging a latch that was never grabbed	Add log phase if validation succeeds (Section 4.3.1.4)

Table 5.1: Three categories of bugs found in Baseline and Pandora: online-failure-free (C1), online-recovery (C2) and recovery (C3).

5.3 Litmus Tests, Bugs, and Fixes

In this section, we discuss our litmus tests in detail and the bugs we have found in both Baseline and Pandora using our litmus testing framework.

In order to pinpoint where the bugs are, we classify the actions of each of these protocols into three distinct categories: online-failure-free (C1), online-recovery (C2), and recovery (C3). Recall, that Baseline inherits C1 and C2 from FORD and C3 from Pandora. Pandora also inherits C1 from FORD. Table 5.1 summarizes and classifies the bugs that we have found, listing the protocol where we found the bug (Baseline and/or Pandora) and the corresponding category.

5.3.1 Litmus 1

Litmus 1 checks direct-write cycles (or violations) between two transactions. As we discussed above, there are two transactions in the litmus test, with the first transaction assigning value V1 to objects X and Y, and the second transaction assigning value V2 to the same two objects. We then assert that the two objects have the same value. Different values imply a strict serializability violation. We also ran variants of this litmus test, replacing writes with inserts and deletes.

Bug: Complicit Abort In this bug, FORD releases every latch in its write-set when it decides to abort. As a result, it also releases some latches that were never actually acquired by the transaction during execution. Crucially, this bug can cause a transaction to release a latch grabbed by a different transaction. Note that this is an online-failure-free (C1) bug that affects both the Baseline and Pandora, because the bug is present in FORD.

Fix: We fix this bug by releasing only the latches that have been actually acquired during execution.

Bug: Missing Actions Additionally, we have found a bug in FORD because logging is omitted for inserts. This is an online-recovery (C2) bug in FORD that affects only the baseline.

Fix: We fix the bug by adding undo logs for inserts (in addition to writes, deletes, and updates).

5.3.2 Litmus 2

Litmus 2 checks read-write Cycles (Figure 5.1(b)). Transaction T1 reads the value of X while updating the value of Y, and T2 reads Y while updating X. Let us assume that T1 reads the old value X=0 and writes Y=1. Since T1 does not see the write of T2, it must be that T2 sees the write of T1. Specifically, if T1 reads X=0 then T2 must read that Y=1. If t2 reads Y=0 and proceeds to commit X=1, the final outcome would be X=1, Y=1 which violates (strict) serializability [189, 29, 53].

Bug: Covert Latches In its validation phase, FORD does not check if the read objects are latched. Recall that FORD checks versions of all the read-only objects in the validation phase. However, it must also ensure that the objects are not latched. Specifically what happens in the litmus test is that transactions T1 and T2 concurrently read X=0 and Y=0 and then latch Y and X. Because the transaction protocol only checks the version numbers during the validation phase, without considering whether they have been latched, both T1 and T2 can progress, leaving objects in an inconsistent state (X=1, Y=1).

Fix: In order to resolve this bug, we fetch both the latch value and version for each read-only object in a single round trip. This is possible because the latch and version for each object in FORD's KVS are stored together. Then, in the validation phase, before comparing versions, we check whether the object is latched; if the object is latched, we abort the transaction.

Bug: Relaxed Latches Litmus Test 2 revealed also another online-failure-free (C1) bug in FORD, where in rare cases validation starts before ensuring all latches have been grabbed. As a result, the execution phase overlaps with the validation phase. This affects both Baseline and Pandora.

Fix: We fix this bug by enforcing that validation happens strictly after latching.

5.3.3 Litmus 3

We use litmus 3 to check multi-variable indirect-write cycles (Figure 5.1(c)). Transaction T1 reads and increments X, and write the new X into Y. T2 reads and increments X but writes the new X into Z. Therefore, at any given time, the values of Y and Z cannot be larger than the value of X; this is checked by the assertions.

Bug: Lost Decision As we discussed in Section 4.3.1.3, FORD writes logs for transactions that may later get aborted. Crucially, it may be impossible for the recovery protocol to tell whether the transaction has aborted, or it has updated all of its objects and thus must be rolled forward. In the litmus test, T1 logs the writes to X and Y but it aborts. The bug occurs when the recovery protocol reads the log and infers that the write to X has been applied because it sees that X has been modified. But the modification has been done by T2, not T1. Because Y has not been modified, the recovery protocol rolls back the update to X. In doing so, it partially undoes T2 and leaves memory in an inconsistent state, as Z has been updated by T2.

Fix: As discussed in Section 4.3.1.4, we add a logging phase after validation, which is executed only if validation succeeds.

Bug: Logging without latching This bug is caused by the problem discussed above (logging and then aborting), in conjunction with a corner case in FORD where a log is written before the latch is actually grabbed. Similarly, with the above, the recovery protocol can either erroneously roll-forward or undo the write of another transaction.

Fix: The logging phase after validation suffices to solve both issues, as it ensures we log after latching. Alternatively, Pandora can enforce latch-to-log order, eliminating the additional round trip.

5.3.4 More Bugs and Fixes

Apart from the above reported bugs, we also came across numerous bugs while testing Pandora's proposed C2 and C3 phases. We have fixed all of those bugs. The next performance evaluation section refers to the fixed versions of the Baseline and Pandora.

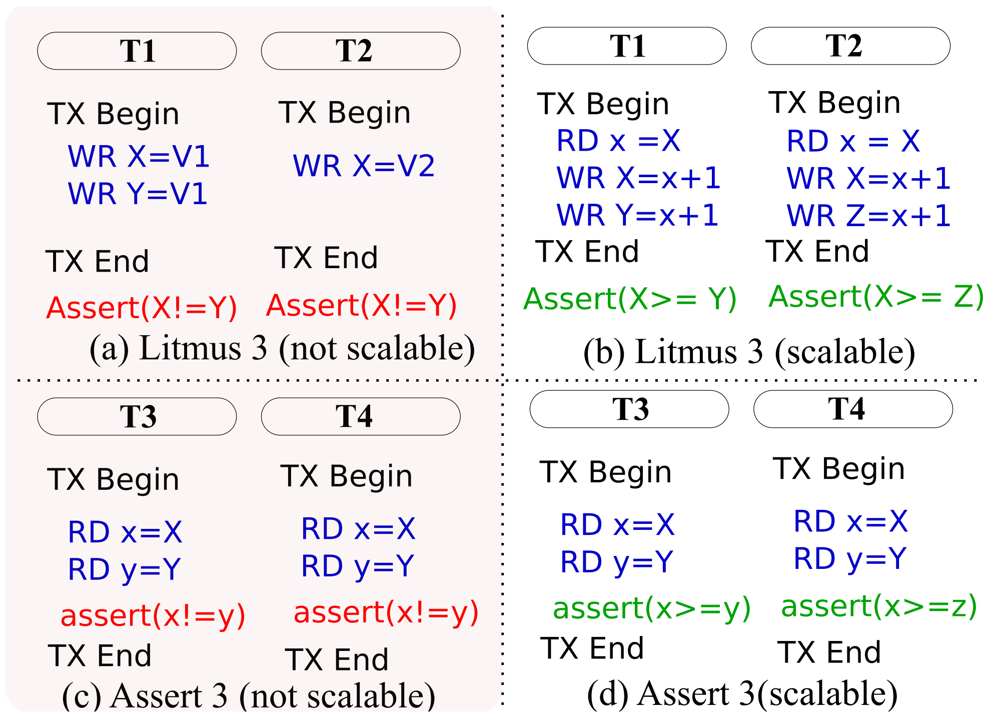


Figure 5.2: Scaling tests and their asserts: scalable and non-scalable asserts

5.4 Test Coverage

Comprehensive test coverage is essential for any testing framework to thoroughly validate system correctness while improving test confidence. DART maximizes coverage in three key ways:

First, non-assert transactions in each DART test can execute recurrently without affecting the test effectiveness. Supporting recurring test execution increases the probability of exposing bugs, as some of the bugs can only be uncovered by precise interleaving of the protocol steps.

Second, scalability plays a pivotal role in achieving comprehensive coverage. DART's advantage lies in separating specification testing from low-level protocol implementation details. This offers two key benefits: It removes randomness from test generation, simplifying scalability through consistency-based test creation, and facilitates scalability of the testing against diverse parameters such as component count and system failures. As a result, our tests seamlessly scale to numerous application instances, with the assert never failing when execution is correct. This expansive coverage surfaces bugs that may manifest only in larger deployments. Figure 5.2 illustrates this: the left test (a and c) is not scalable due to assert failures, while the right test (b and d) is scalable to any number of instances. DART only chooses the latter test.

T1	T2	T3	Assert
TX Begin	TX Begin	TX Begin	TX Begin
WR X=V1	WR X=V2	WR X=V3	RD x=X
WR Y=V1	WR Y=V2	WR Y=V3	RD y=Y
WR Z=V1	WR Z=V2	WR Z=V3	RD z=Z
TX End	TX End	TX End	assert(x=y=y)
Assert(X=Y=Z)		Assert(X=Y=Z)	TX End

(a) litmus 1 and assert (extended-vertical)

T1	T2	T3	Assert
TX Begin	TX Begin	TX Begin	TX Begin
RD x=X	RD y=Y	RD z=Z	RD x=Z
WR Y=x+1	RD Z=y+1	RD X=z+1	RD y=Y
			RD z=Z
TX End	TX End	TX End	assert(x!=y!=z)
Assert(X!=Y!=Z)		Assert(X!=Y!=Z)	TX End

(c) Litmus 2 and Assert (extended-horizontal)

Figure 5.3: Extending tests by changing number of variables

Third, DART facilitates comprehensive test coverage through extensible testing within our framework, a pivotal aspect of end-to-end testing. Each test can be readily expanded both vertically and horizontally. Figure 5.3 illustrates the extension of our foundational tests, Litmus 1 (a) and Litmus 2 (b), incorporating additional variables to augment their efficacy. This extensibility also applies to the inclusion of further asserts within the tests, and it can be applied to other test scenarios.

In summary, factors such as test recurrence, scalable generation, and extensible tests enable DART to maximize coverage by thoroughly exercising diverse execution scenarios. This helps DART ensure the end-to-end correctness of complex transaction protocols.

5.5 Summary

In this chapter, we proposed DART, a novel target litmus-testing framework for validating the correctness of transaction protocols. Unlike state-of-the-art transaction validation frameworks, DART operates as a black-box tester. As such, it does not impose a heavy programming burden and is both scalable and efficient.

One takeaway from this research emphasizes the importance of litmus testing in protocol design. We argue that the design of RDMA-based transaction protocols should incorporate a robust end-to-end testing infrastructure within the design loop. This ensures that any protocol bugs can be iteratively identified and corrected. DART is specifically tailored for this purpose.

Chapter 6

Conclusion

In this section, we discuss the conclusion of this work(§6.1), a critical analysis and limitations(§6.2), lessons learned (§6.3), and future work(§6.4) of this thesis.

6.1 Summary

Taming the growing failures in modern datacenters necessitates fast recovery to minimize interruptions for online users (Availability). However, introducing additional measures to implement recovery negatively impacts performance, correctness, and programmability. Firstly, these recovery measures consume CPU cycles that could otherwise be allocated to handling more client requests in the datacenter (Performance). Secondly, these additional recovery measures make it significantly challenging to achieve end-to-end correctness (Consistency) and maintain programmability in practice.

This thesis leveraged the potential of two emerging memory technologies– specifically non-volatile memory (NVM) and disaggregated memory (DM)– to achieve fast, correct, and performant recovery in the datacenters. Specifically, we showed that emerging memory offers a surprising opportunity for enabling fast recovery with minimal performance overhead. However, both NVM and DM have one crucial problem in common: they expose critical low-level ordering to the high-level software interface, introducing new semantic challenges that can jeopardize correctness and programmability. Therefore, when architecting datacenters with emerging memory, the system architect must, in turn, specify precise consistency models and provide low-level primitives to efficiently enforce them (these models also play a crucial role in optimizing performance opportunities). We addressed these challenges with three contributions: LRP, Pandora, and DART.

First, we demonstrated that non-volatile memory (NVM) enables the opportunity for recovering log-free data structures (LFDs) with minimal performance overhead. The recovery of LFDs, however, hinges on primitives that control the order of updates to NVM during execution; Overlooking this ordering leads to correctness issues, while excessive ordering can significantly impact performance, hindering the opportunity for efficient recovery To enforce correct and minimal ordering, we proposed a new persistency model – Release Persistency (RP) – together with an efficient microarchitectural mechanism, dubbed Lazy Release Persistency (LRP). Our experiments reveal that LRP enables fast and correct recovery for LFDs, surpassing state-of-the-art persistency models by up to 55% in terms of performance improvement. Importantly, RP does not compromise programmability, as its release-acquire semantics align with the well-established DRF-SC (”Sequential Consistency for Data Race Free”) programming model widely used in popular languages like C++ and Java.

Second, we argued that hardware-disaggregated memory (DM) offers a promising yet challenging path to achieving high availability in key-value data stores. High availability hinges on transaction protocols that enable fast and correct recovery. However, traditional transaction protocols that rely on RPCs fail to work with DM, necessitating one-sided accesses. We demonstrated that one-sided accesses expose critical low-level ordering to the high-level software interface, thereby potentially leading to recovery bugs and prohibitively long recovery time. To address these challenges, we proposed Pandora, which provides a fast, correct, and recoverable one-sided transaction protocol. Our implementation artifacts showed that Pandora achieves fast (and correct) recovery within the range of a few milliseconds, which is multiple orders of magnitude faster than state-of-the-art one-sided transaction protocols, all while incurring minimal overhead on performance. Additionally, adapting transactions as our programming model did not compromise on programmability, as transactions have been the de facto programming model in data stores.

Finally, we proposed DART, a target litmus-testing framework for validating the end-to-end correctness of transaction protocols with recovery. We highlighted several critical bugs that DART found in Pandora which we used to iteratively fix the protocol. Crucially, DART is effective and lightweight and can be used as an alternative to model checking which is notoriously hard to verify for transaction protocols. Furthermore, DART serves as a black-box testing method, requiring no additional programming effort, thereby enhancing its utility and applicability for rigorous protocol testing.

In hindsight, this thesis, named TANDEM, successfully achieved the initial goals we

set at its inception. It substantiates our argument that emerging memory technologies can indeed be leveraged to enable fast, correct, and performant recovery in modern datacenters, provided the associated challenges are appropriately addressed. This thesis highlighted the importance of a holistic consistency-driven approach to addressing these challenges.

6.2 Critical Analysis

In this section, we critically analyze our approach and acknowledge the limitations of this thesis.

Traditionally, systems architects ensure end-to-end correctness using consistency models. These models are also crucial for improving programmability and optimizing performance. While these models are well-established, enforcing them in practice with *emerging memory*, especially in the context of failures and recovery, introduces new semantic challenges that pose a significant threat to correctness.

We tackled this challenge by distilling minimal correctness specifications for selected applications, such as log-free data structures (LFDs) and key-value data stores (KVSes). We then focused on two widely used consistency models in the literature: release consistency (for LFDs) and transactions (for KVSes). We subsequently studied these models to account for specific failure scenarios in NVM and DM. To enforce consistency semantics, we proposed two low-level protocols and relied on testing to confirm their adherence to the prescribed consistency model. While practical, this approach lacks formal rigor.

First, one of the key limitations of our approach is the absence of formal proof of correctness guarantees. Instead, we rely on informal proofs wherever possible to validate the correctness of our proposed solutions. While informal proofs and litmus testing can provide valuable insights into the behavior of our systems, they may not offer the same level of rigor and certainty as formal proofs. This limitation implies that there might be cases or edge scenarios where the correctness of our solutions is not fully established.

Furthermore, the lack of formal proofs limits our discussion to only a few selected protocols in each emerging memory category. This limitation is particularly significant when dealing with one-sided transaction protocols in disaggregated memory key-value stores due to their complexity and non-trivial nature. With the formal proofs, we could have generated different one-sided transaction protocols with distinct trade-offs.

Formal proof of correctness is a rigorous mathematical method that provides a strong assurance of the desired properties and behavior of a system. By not having formal proofs, we introduce an element of uncertainty in the correctness claims of our proposed solutions, potentially leaving some scenarios unexplored or not fully validated. However, this is a common problem: formally verifying consistency protocols, especially in the context of failures, is notoriously hard. Not to mention that failures further exacerbate the complexity of this problem. Therefore, further research and development might be required to provide more comprehensive formal proofs for our solutions to enhance their robustness and reliability.

Second, as we have already emphasized, due to the complexity of consistency, we limited our thesis to two basic applications: LFDs and Transactional KVSes [201, 70, 118, 67, 154, 106, 178, 199, 59, 120]. Part of this limitation stems from the lack of formalism because we had to manually derive these protocols and implement them. However, it is crucial to recognize that these applications, while not the only applications running in the datacenter, are an important class of application in the datacenters. The applicability of our proposed solutions may vary when dealing with other types of applications and consistency models, and additional research would be needed to explore their effectiveness in different contexts. However, such applications only account for a small proportion of the datacenter.

The limitation of not providing formal proof of correctness and the narrow scope of our study are important considerations that warrant further attention in future research. Despite these limitations, we believe our work contributes valuable insights into leveraging emerging memory technologies for fault tolerance and performance in datacenters. As we continue to refine and extend our research, a focus on incorporating more formal methods of verification and proof, as well as exploring broader application scenarios, will be essential to strengthen the credibility and impact of our work.

Furthermore, as with any research, there are inherent constraints that we encountered during the course of this thesis. The scope of emerging memory technologies is continually evolving, and while we have focused on non-volatile and disaggregated memory, there may be other emerging memory technologies that could have relevance to fast recovery and performance in datacenters. Our study was limited to the technologies available up to the time of this research, and as newer technologies emerge, there may be additional opportunities and challenges to explore.

In hindsight, while our thesis has made significant strides in addressing challenges in new memory technologies, it is essential to acknowledge the specific limitations

we encountered. The focus on specific applications, the evolving nature of emerging memory technologies, the absence of formal proof of correctness, and the potential variations in applicability to other datacenter applications all contribute to the scope of our work. As we navigate these limitations and continue to expand our understanding, future research and exploration will play a vital role in advancing the application of emerging memory technologies in the broader context of datacenter architecture.

6.3 Lessons Learned

In this section, we discuss two lessons that we learned while working on this thesis. These insights are important for future research in this direction.

6.3.1 Lesson 1: Formal Methods vs Testing

The most precise approach to validate correctness is by formal methods, which constitute a set of rigorous mathematically based techniques for specifying, designing, and verifying software and hardware systems. These methods involve using formal languages and mathematical logic to describe system behaviors, properties, and specifications. Formal methods encompass a range of techniques, including model checking, theorem proving, and static analysis.

For example, model checking is geared towards exhaustive verification of whether a provided system model satisfies a desired property. In the process of model checking, a formal representation of the system's behavior is constructed, often utilizing finite state machines or temporal logic. The model checker systematically traverses through all potential states and transitions within the model, aiming to confirm the validity of the desired property or identify any counterexamples.

Formally verifying distributed systems, while necessary, is notoriously hard [45, 173, 103, 214, 151, 213]. Specifically, in distributed systems, expressing behavior mathematically is extremely hard as it requires more effort and lines of codes than designing the system. Additionally, verifying abstract specifications or models, due to low-level complexities like failures, does not always guarantee that the actual implementation is correct. An alternative approach to ensure correctness is thorough testing [147, 141, 148, 161, 89, 74].

Testing exhausts the system with possible inputs to see if the systems behave differently than intended properties [147, 115]. Testing, while effective in validating

both specifications and implementation, has a lower chance of finding correctness issues. Actually, it depends on the effectiveness of testing [142, 98, 197, 46, 137]. Typically, system designers have used random testing which is not sufficient.

We have learned neither approach, formal methods nor random testing, is desirable in the datacenter. In this thesis, we have adopted a hybrid approach for testing where we derive test cases from the correctness properties (consistency models). Such methods can further be strengthened with formal methods which we leave as future work [206]. Similarly, system architects can adopt hybrid techniques to reduce the complexity of techniques like model checking.

6.3.2 Lesson 2: Inherent Complexity of Consistency

Consistency models have been studied for decades. However, models like linearizability or sequential consistency, while intuitive and prevalent in many systems, lack sufficient definitions for handling failures and recovery. For instance, linearizability, a widely considered standard in many systems, is not defined under failures, introducing ambiguity for architects and programmers. (Additionally, these models do not say anything about the behavior of transactions [105].) Therefore, when architecting datacenters for failures, architects must pay careful attention to the consistency models.

Conversely, distributed systems experts turn to consistency models like strict serializability to precisely outline correctness in failure-prone scenarios [187]. Nevertheless, these models lack formal definitions and can lead to confusion, especially due to their similarity to terms like serializability commonly used in database isolation. In an effort to mitigate these confusions, architects have introduced new terminology such as Strong Consistency or External Consistency [52]. However, this approach has inadvertently compounded the issue, given that the same term is also used to contrast with more relaxed consistency models like eventual consistency [17, 16].

Similarly, the database community relies on ACID to describe end-to-end consistency, arguably the most rigorously defined model in the presence of failures [30, 31]. Nonetheless, the ACID properties—Atomicity, Consistency, Isolation, Durability—often bewilder architects and programmers [104, 127, 53].

These intricate nuances of consistency become evident when designing datacenters with emerging memory technologies. For example, to address the challenges of linearizability, architects have recently proposed novel consistency models such as *durable linearizability* or *detectability* [73, 22, 143, 48, 162], which remain active areas

of research. The crux of the matter is that precise consistency semantics are crucial when architecting datacenters with new technologies, despite the inherent challenges stemming from complexity and ambiguity. However, contrary to common belief, we have realized that there are still issues with consistency definitions (formal models).

6.4 Future Work

While this thesis has made significant progress in addressing the challenges of emerging memory technologies, several avenues for future research and exploration remain open. The following areas represent potential directions for further investigation and development:

6.4.1 Formal Proof

One of the primary limitations of our approach is the lack of formal proof of correctness guarantees for the proposed solutions. Although we have utilized informal proofs and litmus testing to validate the correctness and recovery of our systems, the adoption of formal methods offers a higher level of rigor and certainty. Future work should focus on developing formal verification techniques specifically tailored for consistency protocols, especially within the context of transactions. Enhancing the robustness and reliability of our solutions through formal proofs would significantly strengthen their credibility and applicability in real-world datacenter deployments.

6.4.2 Verifiably Correct Transaction Protocols

Fault-tolerant transactions stand as the most important consistency model within datacenters. Yet, the design and verification of transaction protocols remain notoriously hard. Using ad-hoc techniques and trial-and-error testing is not sufficient. The need for correct and recoverable one-sided transaction protocol in the future will exacerbate this problem. Can we systematically design transaction protocols that are verifiably correct? One insight from this thesis is that there is a possibility to design and verify transaction protocols by formally modeling their correctness properties including recovery (i.e. consistency models). Using these models we can synthesize transaction protocols while formally verifying their correctness properties. Rigorous testing like DART proposed in this thesis can be further developed to improve the confidence of such an approach. We will investigate this aspect of the protocols in the future.

6.4.3 Incorporating New Emerging Memory Technologies

The field of emerging memory technologies is continuously evolving, and new technologies may emerge after the completion of this thesis. Future work should keep abreast of the latest developments in emerging memory and assess their relevance to fault tolerance and performance in datacenters. Exploring the strengths and limitations of novel emerging memory technologies will offer fresh insights into potential opportunities and challenges that were not within the scope of this research. For instance, looking into how new disaggregated memory hardware like Compute eXpress Links (CXL) impacts our design decision is an interesting research avenue [139, 90, 183].

6.4.4 Final Remarks

In conclusion, while this thesis has laid a strong foundation in addressing challenges of fast, correct, and performant recovery in modern datacenters with the use of emerging memory technologies, there are exciting opportunities for future research and development. By focusing on formal verification, synthesizing transaction protocol, and keeping up with evolving technologies, we can continue to advance the application of emerging memory technologies in the broader context of datacenter architecture. The pursuit of these future research directions will undoubtedly contribute to the ongoing advancement of datacenter technology and its pivotal role in supporting our rapidly evolving cloud ecosystem.

Bibliography

[1] https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[2] Tatp. telecom application transaction processing benchmark, 2011.

[3] Smallbank benchmark, 2021.

[4] Tpc-c benchmark, 2021.

[5] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[6] Sarita V Adve and Mark D Hill. Weak ordering—a new definition. *ACM SIGARCH Computer Architecture News*, 18(2SI):2–14, 1990.

[7] Atul Adya. Weak consistency: a generalized theory and optimistic implementations for distributed transactions. 1999.

[8] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, 1999.

[9] Marcos K. Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. Memory disaggregation: Why now and what are the challenges. *SIGOPS Oper. Syst. Rev.*, 57(1):38–46, jun 2023.

[10] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. Remote regions: a simple abstraction for remote memory. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 775–787, 2018.

[11] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 120–126, 2019.

[12] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 120–126, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. Springer.

[14] Peter Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *International Conference on Software Engineering*, 1976.

[15] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[16] Amazon. Amazon s3 strong consistency, 2020. [Online; accessed 8-September-2023].

[17] Amazon. Amazon s3 update – strong read-after-write consistency, 2020. [Online; accessed 8-September-2023].

[18] ARM Limited. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*, 10 2018. Initial v8.4 EAC release.

[19] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[20] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. Technical report, Technical Report UCB/EECS-2009-28, EECS Department, University of California . . . , 2009.

[21] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-passing Systems. *J. ACM*, 42(1):124–142, January 1995.

[22] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Detectable recovery of lock-free data structures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 262–277, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 487–498, New York, NY, USA, 2011. ACM.

[24] Peter Bailis and Kyle Kingsbury. The network is reliable. *Commun. ACM*, 57(9):48–55, sep 2014.

[25] Richard Barker and Paul Massiglia. *Storage Area Network Essentials: A Complete Guide to Understanding and Implementing SANs*. Wiley Publishing, 1st edition, 2001.

[26] Luiz Andr Barroso, Jimmy Clidaras, and Urs Hlzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2nd edition, 2013.

[27] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The datacenter as a computer: Designing warehouse-scale machines*. Springer Nature, 2019.

[28] Ali Basiri, Niosha Behnam, Ruud De Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[29] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[30] Philip A Bernstein et al. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley Reading, 1987.

[31] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, (3):203–216, 1979.

[32] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[33] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *arXiv preprint arXiv:1504.01048*, 2015.

[34] Guy E. Blelloch, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The Parallel Persistent Memory Model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 247–258, New York, NY, USA, 2018. ACM.

[35] Hans-J. Boehm. Can seqlocks get along with programming language memory models? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '12, pages 12–20, New York, NY, USA, 2012. ACM.

[36] Hans-J Boehm and Sarita V Adve. Foundations of the c++ concurrency memory model. *ACM SIGPLAN Notices*, 43(6):68–78, 2008.

[37] Hans-J Boehm and Dhruva R Chakrabarti. Persistence programming models for non-volatile memory. *ACM SIGPLAN Notices*, 51(11):55–67, 2016.

[38] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on s3. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 251–264, New York, NY, USA, 2008. Association for Computing Machinery.

[39] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. 1993.

[40] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[41] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. Durable functions: Semantics for stateful serverless. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27, 2021.

[42] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 79–92, 2021.

[43] Josiah Carlson. *Redis in action*. Simon and Schuster, 2013.

[44] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[45] Bo-Shoe Chen and R.T. Yeh. Formal specification and verification of distributed systems. *IEEE Transactions on Software Engineering*, SE-9(6):710–722, 1983.

[46] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. Cofi: Consistency-guided fault injection for cloud systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 536–547, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Brian Cho and Ergin Seyfe. Taking advantage of a disaggregated storage and compute architecture. *Spark+ AI Summit*, 2019.

[48] Kyeongmin Cho, Seungmin Jeon, and Jeehoon Kang. Practical detectability for persistent lock-free data structures. *arXiv preprint arXiv:2203.07621*, 2022.

[49] Tzvi Chumash. Obtaining five "nines" of availability for internet services. *Rutgers University*, 2019, 2005.

[50] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of*

the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[51] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[52] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[53] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is believing: A client-centric specification of database isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 73–82, New York, NY, USA, 2017. Association for Computing Machinery.

[54] Mahesh Dananjaya. Release persistency, 2019. Available at https://project-archive.inf.ed.ac.uk/msc/20193543/msc_proj.pdf.

[55] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. Lazy release persistency. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1173–1186, 2020.

[56] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 373–386, Boston, MA, July 2018. USENIX Association.

[57] Edsger W. Dijkstra. *Self-Stabilizing Systems in Spite of Distributed Control*, page 333–338. Association for Computing Machinery, New York, NY, USA, 1 edition, 2022.

[58] diy. diy. https://diy.inria.fr/, 2020. [Online; accessed 19-July-2022].

[59] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[60] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 54–70, New York, NY, USA, 2015. Association for Computing Machinery.

[61] Jingwen Du, Fang Wang, Dan Feng, Changchen Gan, Yuchao Cao, Xiaomin Zou, and Fan Li. Fast one-sided rdma-based state machine replication for disaggregated memory. *ACM Trans. Archit. Code Optim.*, mar 2023.

[62] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 1–14, USA, 2019. USENIX Association.

[63] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[64] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, apr 1988.

[65] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 215–229, 1985.

[66] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A scalable, predictably

performant, and fully managed NoSQL database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1037–1048, Carlsbad, CA, July 2022. USENIX Association.

[67] Avner Elizarov, Guy Golan-Gueta, and Erez Petrank. Loft: Lock-free transactional data structures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 425–426, New York, NY, USA, 2019. Association for Computing Machinery.

[68] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 299–313. ACM, 2015.

[69] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[70] Daniel Gomez Ferro, Flavio Junqueira, Ivan Kelly, Benjamin Reed, and Maysam Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *2014 IEEE 30th International Conference on Data Engineering*, pages 676–687. IEEE, 2014.

[71] Armando Fox and Eric A Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 174–178. IEEE, 1999.

[72] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A Persistent Lock-free Queue for Non-volatile Memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 28–40, New York, NY, USA, 2018. ACM.

[73] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. *ACM SIGPLAN Notices*, 53(1):28–40, 2018.

[74] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 100–115, 2021.

[75] Yaosheng Fu and David Wentzlaff. PriME: A parallel and distributed simulator for thousand-core chips. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pages 116–125, 2014.

[76] Vasilis Gavrielatos. Designing the replication layer of a general-purpose datacenter key-value store. 2021.

[77] Vasilis Gavrielatos, Antonios Katsarakis, and Vijay Nagarajan. Odyssey: The impact of modern hardware on strongly-consistent replication protocols. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 245–260, 2021.

[78] Vasilis Gavrielatos, Antonios Katsarakis, Vijay Nagarajan, Boris Grot, and Arpit Joshi. Kite: Efficient and available release consistency for the datacenter. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–16, 2020.

[79] Kourosh Gharachorloo. *Memory consistency models for shared-memory multiprocessors*. Stanford University, 1996.

[80] Kourosh Gharachorloo, Sarita V Adve, Anoop Gupta, John L Hennessy, and Mark D Hill. Programming for different memory consistency models. *Journal of parallel and distributed computing*, 15(4):399–407, 1992.

[81] Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. Two techniques to enhance the performance of memory consistency models. In *ICPP (1)*, pages 355–364, 1991.

[82] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *ACM SIGARCH Computer Architecture News*, 18(2SI):15–26, 1990.

[83] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[84] Garth A. Gibson and Rodney Van Meter. Network attached storage architecture. *Commun. ACM*, 43(11):37–45, nov 2000.

[85] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 350–361, 2011.

[86] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for Synchronization-free Regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 46–61, New York, NY, USA, 2018. ACM.

[87] Vaibhav Gogte, Aasheesh Kolli, and Thomas F Wenisch. *A primer on memory persistency*. Morgan & Claypool Publishers, 2022.

[88] Vaibhav Gogte, Aasheesh Kolli, and Thomas F Wenisch. *A primer on memory persistency*. Morgan & Claypool Publishers, 2022.

[89] João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. Mumak: Efficient and black-box bug detection for persistent memory. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 734–750, 2023.

[90] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access,{High-Performance} memory disaggregation with {DirectCXL}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, 2022.

[91] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access,{High-Performance} memory disaggregation with {DirectCXL}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, 2022.

[92] Vincent Gramoli. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 1–10, New York, NY, USA, 2015. ACM.

[93] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[94] Steven D Gribble, Eric A Brewer, and Joseph M Hellerstein. Scalable, distributed data structures for internet service construction. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[95] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. {uKharon}: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 101–120, 2022.

[96] Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In *International conference on reliable software technologies*, pages 38–57. Springer, 1996.

[97] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[98] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM symposium on cloud computing*, pages 1–14, 2014.

[99] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 417–433, New York, NY, USA, 2022. Association for Computing Machinery.

[100] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*, 15(4):287–317, 1983.

[101] Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[102] Tamás Hauer, Philipp Hoffmann, John Lunney, Dan Ardelean, and Amer Diwan. Meaningful availability. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 545–557, 2020.

[103] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, 2015.

[104] Pat Helland. Acid: My personal: How could i miss such a simple thing? *Queue*, 19(2):17–20, jun 2021.

[105] Pat Helland. Don't get stuck in the" con" game: Consistency, convergence, and confluence are not the same! eventual consistency and eventual convergence aren't the same as confluence, either. *Queue*, 19(3):16–35, 2021.

[106] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, 1993.

[107] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[108] 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.

[109] Leah Hoffmann. 'everything fails all the time'. *Commun. ACM*, 63(2):96–ff, jan 2020.

[110] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[111] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, pages 313–327, 2016.

[112] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[113] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR*, abs/1903.05714, 2019.

[114] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26, 2019.

[115] Jepsen. Jepsen. https://jepsen.io/, 2020. [Online; accessed 19-July-2022].

[116] Ricardo Jiménez-Peris, M. Patiño Martínez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, sep 2003.

[117] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 660–671, New York, NY, USA, 2015. ACM.

[118] Flavio Junqueira, Benjamin Reed, and Maysam Yabandeh. Lock-free transactional support for large-scale storage systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 176–181. IEEE, 2011.

[119] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance {RDMA} systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, 2016.

[120] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 185–201, USA, 2016. USENIX Association.

[121] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 654–663, New York, NY, USA, 1997. Association for Computing Machinery.

[122] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. ASPLOS '20, page 201–217, New York, NY, USA, 2020. Association for Computing Machinery.

[123] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[124] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. *SIGARCH Comput. Archit. News*, 20(2):13–21, April 1992.

[125] Clay Kelly. Amazon.com goes down, loses 66,240 dollars per minute. *Forbes*.

[126] 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.

[127] Kyle Kingsbury. Jepsen: A framework for distributed systems verification, with fault injection, 2018.

[128] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 481–493, New York, NY, USA, 2017. ACM.

[129] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated Persist Ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 58:1–58:13, Piscataway, NJ, USA, 2016. IEEE Press.

[130] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. Farview: Disaggregated memory with operator off-loading for database engines. *arXiv preprint arXiv:2106.07102*, 2021.

[131] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[132] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, 100(9):690–691, 1979.

[133] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[134] Leslie Lamport. Specifying systems: the tla+ language and tools for hardware and software engineers. 2002.

[135] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313, 2009.

[136] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K Aguilera, Kimberly Keeton, and Vijay Chidambaram. Dinomo: An elastic, scalable, high-performance key-value store for disaggregated persistent memory (extended version). *arXiv preprint arXiv:2209.08743*, 2022.

[137] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S Gunawi, and Shan Lu. Dfix: automatically fixing timing bugs in distributed systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 994–1009, 2019.

[138] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al.

Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[139] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[140] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 467–483, 2016.

[141] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. Pcatch: Automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[142] Jiaxin Li, Yiming Zhang, Shan Lu, Haryadi S Gunawi, Xiaohui Gu, Feng Huang, and Dongsheng Li. Performance bug analysis and detection for distributed storage and computing systems. *ACM Transactions on Storage*, 19(3):1–33, 2023.

[143] Nan Li and Wojciech Golab. Detectable Sequential Specifications for Recoverable Shared Objects. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[144] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. *SIGARCH Comput. Archit. News*, 37(3):267–278, jun 2009.

[145] 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 International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.

[146] litmus7. litmus7. https://diy.inria.fr/doc/litmus.html, 2020. [Online; accessed 19-July-2022].

[147] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. *ACM SIGARCH Computer Architecture News*, 45(1):677–691, 2017.

[148] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. Fcatch: Automatically detecting time-of-fault bugs in cloud systems. *ACM SIGPLAN Notices*, 53(2):419–431, 2018.

[149] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, Santa Clara, CA, February 2020. USENIX Association.

[150] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[151] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 370–384, 2019.

[152] Rupak Majumdar and Filip Niksic. Why is random testing effective for partition tolerance bugs? *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[153] Jeremy Manson, William Pugh, and Sarita V Adve. The java memory model. *ACM SIGPLAN Notices*, 40(1):378–391, 2005.

[154] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.

[155] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. Ibm storage tank—a heterogeneous scalable san file system. *IBM Systems Journal*, 42(2):250–267, 2003.

[156] Maged M. Michael. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[157] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[158] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided {RDMA} reads to build a fast, cpu-efficient key-value store. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 103–114, 2013.

[159] C Mohan and Frank Levine. Aries/im: an efficient and high concurrency index management method using write-ahead logging. *ACM Sigmod Record*, 21(2):371–380, 1992.

[160] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[161] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding {Crash-Consistency} bugs with bounded {Black-Box} crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 33–50, 2018.

[162] Liad Nahum, Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Recoverable and detectable fetch&add. In *25th International Conference on Principles of*

Distributed Systems (OPODIS 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[163] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 135–148, New York, NY, USA, 2017. ACM.

[164] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *NSDI*, volume 17, pages 17–33, 2017.

[165] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–410, 2012.

[166] Dushyanth Narayanan and Orion Hodson. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 401–410, New York, NY, USA, 2012. ACM.

[167] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 317–328, New York, NY, USA, 2014. ACM.

[168] Faisal Nawab, Dhruva R. Chakrabarti, Terence Kelly, and Charles B. Morrey III. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 689–694, 2015.

[169] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A Periodically Persistent Hash Map. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[170] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[171] Open Compute Project. Open compute project, 2020. [Online; accessed 8-September-2023].

[172] Open19. Open19, 2020. [Online; accessed 8-September-2023].

[173] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 614–630, 2016.

[174] Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[175] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. *ACM SIGARCH Computer Architecture News*, 42(3):265–276, 2014.

[176] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.

[177] Arash Pourhabibi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. Cerebros: Evading the rpc tax in datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 407–420, New York, NY, USA, 2021. Association for Computing Machinery.

[178] Ravi Rajwar and James R Goodman. Transactional lock-free execution of lock-based programs. *ACM SIGOPS Operating Systems Review*, 36(5):5–17, 2002.

[179] Ganesan Ramalingam and Kapil Vaswani. Fault tolerance via idempotence. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 249–262, 2013.

[180] Kurt Rothermel and C Mohan. *ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions.* Citeseer, 1989.

[181] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. Aifm: High-performance, application-integrated far memory. In *Proceedings of*

the 14th USENIX Conference on Operating Systems Design and Implementation, pages 315–332, 2020.

[182] Thomas M Ruwart. Osd: a tutorial on object storage devices. In *NASA CONFERENCE PUBLICATION*, pages 21–34. Citeseer, 2002.

[183] Samsung. Samsung develops industry's first cxl dram supporting cxl 2.0. *Samsung Newsroom*.

[184] Richard D Schlichting and Fred B Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3):222–238, 1983.

[185] Michael L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, 2013.

[186] Yasas Seneviratne, Korakit Seemakhupt, Sihang Liu, and Samira Khan. Nearpm: A near-data processing system for storage-class applications. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 751–767, New York, NY, USA, 2023. Association for Computing Machinery.

[187] Ravi Sethi. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM (JACM)*, 29(2):394–403, 1982.

[188] Ravi Sethi. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM (JACM)*, 29(2):394–403, 1982.

[189] Ravi Sethi. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM (JACM)*, 29(2):394–403, 1982.

[190] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed os for hardware resource disaggregation. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 69–87, USA, 2018. USENIX Association.

[191] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. Towards a fully disaggregated and programmable data center. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 18–28, 2022.

[192] Sijie Shen, Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. Drtm+b: Replication-driven live reconfiguration for fast and general distributed transaction

processing. *IEEE Transactions on Parallel and Distributed Systems*, 33(10):2628–2643, 2022.

[193] Dale Skeen. A quorum-based commit protocol. Technical report, Cornell University, 1982.

[194] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, (3):219–228, 1983.

[195] James W Stamos and Flaviu Cristian. Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1:383–408, 1993.

[196] Ion Stoica and Scott Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 26–32, 2021.

[197] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. Automatic reliability testing for cluster management controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 143–159, Carlsbad, CA, July 2022. USENIX Association.

[198] Jeff Terrace and Michael J Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference*, 2009.

[199] TiKV. Tikv, 2023. [Online; accessed 8-September-2023].

[200] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[201] John David Valois. *Lock-free data structures*. Rensselaer Polytechnic Institute, 1995.

[202] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, 2004.

[203] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[204] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[205] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *NSDI*, volume 20, pages 449–462, 2020.

[206] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. Model checking guided testing for distributed systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 127–143, New York, NY, USA, 2023. Association for Computing Machinery.

[207] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1033–1048, New York, NY, USA, 2022. Association for Computing Machinery.

[208] Andrew Waterman, Yunsup Lee, David A. Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. The risc-v instruction set manual, 2014.

[209] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 87–104, New York, NY, USA, 2015. Association for Computing Machinery.

[210] Wikipedia contributors. Hard disk drive — Wikipedia, the free encyclopedia, 2023. [Online; accessed 8-September-2023].

[211] Yi Xu, Joseph Izraelevitz, and Steven Swanson. Clobber-nvm: log less, re-execute more. In *Proceedings of the 26th ACM International Conference on*

Architectural Support for Programming Languages and Operating Systems, pages 346–359, 2021.

[212] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, 2020.

[213] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. {DuoAI}: Fast, automated inference of inductive invariants for verifying distributed protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 485–501, 2022.

[214] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. {DistAI}:{Data-Driven} automated invariant learning for distributed protocols. In *15th USENIX symposium on operating systems design and implementation (OSDI 21)*, pages 405–421, 2021.

[215] Sangho Yi, Artur Andrzejak, and Derrick Kondo. Monetary cost-aware checkpointing and migration on amazon cloud spot instances. *IEEE Transactions on Services Computing*, 5(4):512–524, 2011.

[216] Yifan Yuan, Jinghan Huang, Yan Sun, Tianchen Wang, Jacob Nelson, Dan RK Ports, Yipeng Wang, Ren Wang, Charlie Tai, and Nam Sung Kim. Rambda: Rdma-driven acceleration framework for memory-intensive μs-scale datacenter applications. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 499–515. IEEE, 2023.

[217] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. The end of a myth: Distributed transactions can scale. *arXiv preprint arXiv:1607.00655*, 2016.

[218] Deli Zhang and Damian Dechev. An Efficient Lock-Free Logarithmic Search Data Structure Based on Multi-dimensional List. *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 281–292, 2016.

[219] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 51–68, Santa Clara, CA, February 2022. USENIX Association.