Architectural Decoupling in Concurrent Data Structures

Architectural Decoupling in Concurrent Data Structures

by

Yaodong Sheng

Presented to the Graduate and Research Committee
of Lehigh University
in Candidacy for the Degree of
Doctor of Philosophy

in

Computer Science

Lehigh University

January 2026

Dissertation Signature Sheet

This document is approved and recommended for acceptance as a dissertation in fulfillment of the requirements for the degree of Doctor of Philosophy.

Architectural Decoupling in Concurrent Data Structures

Yaodong Sheng

Defense Date

Michael Spear
Associate Professor
Department of Computer Science and Engineering
Dissertation Director

Henry F. Korth
Professor
Department of Computer Science and Engineering
Dissertation Committee

Approval Date

Roberto Palmieri
Associate Professor
Department of Computer Science and Engineering
Dissertation Committee

Trevor Brown
Associate Professor
Department of Computer Science, University of Waterloo
Dissertation Committee

Acknowledgments

I would like to express my deepest gratitude to my advisor, Professor Michael Spear, for his invaluable guidance, patience, and support throughout my doctoral studies. His insights and encouragement have been instrumental in shaping this research and my development as a researcher.

I am grateful to my dissertation committee members for their time, constructive feedback, and valuable suggestions that have significantly improved this work.

I would also like to thank my collaborators and colleagues at Lehigh University for the stimulating discussions and collaborative research that contributed to this dissertation.

Finally, I extend my heartfelt thanks to my family and friends for their unwavering support and encouragement throughout this journey.

Table of Contents

List of Figures

Abstract

Concurrent programming has long been constrained by false dichotomies: programmability versus performance, generality versus efficiency, and composition versus specialization. Practitioners must choose between easy-to-use abstractions with poor performance (software transactional memory, coarse-grained locking) or high-performance implementations requiring heroic effort (hand-tuned lock-free algorithms, specialized non-blocking code). This dissertation challenges this conventional wisdom by demonstrating that these tradeoffs are not fundamental but rather artifacts of inadequate information exposure at layer boundaries.

The core insight is that by identifying and exposing the right semantic invariants—not just APIs, but the exact theoretical properties that enable safe optimization (consistency guarantees, conflict rules, object-metadata mappings)—through stable, architectural contracts, systems can separate what must be correct (synchronization semantics) from what must be efficient (implementation strategies) while enabling incremental optimization. This principle is validated through three systems spanning shared memory, multi-structure composition, and distributed RDMA.

exoTM/STMCAS (shared memory) exposes orec values as proofs of immutability, enabling programmers to checkpoint read-only traversals and safely

transition to write steps without maintaining full transaction logs. Two policies coexist: STM for complex, uncommon paths and STMCAS for performance-critical operations. Evaluation shows STMCAS-based data structures usually outperform hand-tuned baselines, reaching up to 1.35× their throughput while preserving STM programmability where needed.

Harmony (composition) exposes co-located but distinct metadata—operation-level synchronization (temporal ownership, versioning) and transaction-level conflict detection (transactional ownership, semantic/structural versions)—avoiding false conflicts when structural changes are mistaken for semantic conflicts. Three compatible techniques for expressing non-existence enable range queries. Evaluation on TPC-C shows Harmony matches or exceeds Medley, the state-of-the-art transactional data structure library, while supporting complex operations uncommon in prior systems.

RDMASTM (RDMA-based) presents a layered architecture that separates data-structure logic, synchronization semantics, and hardware access through an Access & Placement Substrate (APS). APS sits between data structures and the synchronization layer, exposing pluggable policies—object-level caching, validation optimization, batching, clock management, and placement control—while hiding hardware-specific details. Programmers structure objects to exploit spatial locality: fields placed in the same object benefit from single-operation cache fetch and validation-skip optimization, where the framework internally elides redundant orec checks when accessing multiple co-located fields within cached objects. The readset trimming API (`trim(n)`) provides orthogonal optimization—reducing commit-time validation overhead by 16–30% when data-structure semantics permit safe pruning—by leveraging structure-specific invariants. On CloudLab deployments (1–8

nodes, four data structures, three workloads), object-level caching delivers the largest gains (nearly 2× throughput, 50–80% RDMA reduction) by collapsing pointer-chasing and skipping redundant validations. Clock and batching optimizations show workload-dependent effects: clock variants differ by <20%; batching helps (5–10%) only when memory concentrates and workloads are write-heavy, showing minimal benefit (within ±2–3%) when memory is distributed. The stable field-level API (transactional operations) keeps data-structure algorithm logic unchanged—programmers add structural optimizations like anchors for hot metadata or cache-aware layouts for spatial locality—enabling systematic policy exploration while preserving STM correctness.

Across all three projects, stable contracts enable incremental refinement: Programmers start with correct-by-default implementations and selectively optimize performance-critical layers without rewriting data-structure logic. The overarching contribution is establishing that right-sized architectural decoupling—exposing semantic invariants without leaking implementation details—achieves both programmability and performance across diverse synchronization domains, challenging decades of accepted tradeoffs in concurrent programming.

Chapter 1

Introduction

1.1 Motivation: The Parallelism Imperative

In modern processors, the only way to achieve peak performance is by harnessing parallelism. This fundamental reality shapes how we design software: high-performance applications require data structures that can be accessed simultaneously by many threads without compromising correctness. However, achieving this combination of safety and speed remains one of the most challenging tasks in systems programming.

Since Dijkstra's foundational work in the 1960s [1, 2, 3], we have understood that synchronization—the coordination of multiple threads accessing shared resources—is both essential and difficult. Without proper synchronization, concurrent threads can interleave their operations in ways that lead to race conditions, where the behavior of the system depends on unpredictable timing, resulting in incorrect outcomes. A common solution is to encapsulate shared data in "concurrent data structures" that handle synchronization internally [4, 5, 6, 7], providing familiar interfaces while hiding complex synchronization logic.

Yet implementing these concurrent data structures remains extraordinarily

difficult. Programmers must reason about low-level hardware primitives like atomic instructions, memory barriers, and cache coherency protocols, carefully analyzing all potential interleavings among instructions to prevent errors. The code is complicated and hard to maintain; even expert programmers struggle to use these primitives correctly [8]. Moreover, code relying on hardware-specific primitives faces portability challenges when moving across architectures (e.g., from x86 to ARM), limiting scalability across platforms.

1.2 The Promise and Limitations of Transactional Memory

Transactional Memory (TM) emerged as a promising solution to these challenges. Originally proposed as a hardware mechanism (HTM) [9] and later extended to software (STM) [10], TM allows programmers to mark regions of code as transactions that appear to execute atomically. The TM runtime tracks the memory locations each transaction reads and writes (its read-set and write-set); if two transactions conflict—attempting to access the same location with at least one write—the system detects the conflict and aborts one transaction, which is then restarted. This abstraction simplifies reasoning: Each transaction can be treated as if it were a single atomic block, and transactions can be composed into larger transactions [11]. The TM system handles synchronization internally, detecting conflicts and managing retries, significantly reducing the risk of deadlocks and race conditions.

Despite these advantages, TM has not seen widespread adoption because of several fundamental challenges:

Hardware TM offers excellent performance by executing transactions specu-

latively in the CPU's private caches, using the cache coherence protocol to track read- and write-sets. When another thread's access conflicts with a transaction's tracked locations, the coherence protocol detects it and triggers an abort. While fast, HTM is fragile: cache capacity limits restrict transaction size, interrupts or I/O operations can cause spurious aborts, and TSX-related security issues have led some systems to restrict or disable HTM [12].

Software TM provides generality and portability by instrumenting memory accesses to build and manage read- and write-sets in software data structures. On commit, STM validates that no conflicting accesses have occurred before making changes permanent. This flexibility comes at a cost: the instrumentation and metadata management introduce significant overheads, especially in conflict detection and resolution under high contention, leading to frequent aborts that degrade performance. Research has shown that performance improves when STM is optimized for specific data structures or access patterns [13, 14], or when programmers directly manage transaction metadata [15]. In other words, sacrificing generality or ease of use can yield much better performance—but at the cost of the very programmability that made TM attractive in the first place.

In the wake of TM's limited adoption, the field largely bifurcated. One path pursued highly specialized, non-blocking algorithms that, while performant, require near-heroic implementation effort and are notoriously difficult to compose [8]. The other continued to rely on coarse-grained locking, sacrificing performance for simplicity. Both paths reinforce the same difficult trade-off between performance and programmability.

These fundamental trade-offs—between ease of use and performance, between generality and efficiency, and between composition and specialization—have long

been viewed as unavoidable compromises in concurrent programming. Systems either provide simple abstractions with poor performance, expose low-level control at the cost of complexity and fragility, or achieve high performance through non-composable specialization.

1.3 Thesis

This dissertation challenges this status quo. We assume familiarity with concurrency but not with these systems; this section states the core idea and previews the three projects. Rather than accepting fixed trade-offs among ease of use, generality, and performance, we separate the system into a small number of clear layers, each with a specific responsibility and a well-defined service to the layer above. High-level layers provide familiar, general programming abstractions. Low-level layers implement performance-critical mechanisms such as ownership record (orec) tracking—metadata structures that track which transaction owns which object—along with conflict detection and version management. Crucially, the interfaces between layers are *stable*: the northbound semantics—what higher layers can assume (e.g., atomicity/serializability and conflict rules)—do not change when we swap implementations or policies. Because the contract between layers stays the same, we can optimize or replace lower-level code without touching higher-level code. The interfaces also surface the right information (e.g., which conflicts are detected and how orecs map to objects), letting higher layers choose effective optimizations. Together, this layered design enables incremental optimization: developers can start with a simple, correct implementation and progressively tune performance-critical layers as hot paths emerge, without rewriting data-structure logic. This provides a systematic path to high performance through isolated, independently-optimizable

layers.

The insight draws inspiration from a principle well-established in operating systems design [16, 17, 18]: by isolating core functionality from the policies that govern its use, we can build systems that support multiple programming models simultaneously, each optimized for different use cases. At a high level, we separate *how* operations synchronize (ensuring atomicity and ordering) from *how* systems manage transactions and performance (conflict detection, caching, batching). This lets us tailor policies without changing the core programming model. However, applying this principle to fine-grained concurrency has remained elusive, as the concerns of atomicity, conflict management, and performance optimization have been historically intertwined in monolithic implementations. Our key insight is that these concerns can be systematically separated through carefully designed interfaces: minimal mechanisms for tracking ownership and ordering (separating synchronization/atomicity from policy), co-located metadata distinguishing synchronization from conflict detection (operation-transaction separation), or layered abstractions isolating application logic from the communication fabric (application-network separation). Critically, these layers are *right-sized*: they expose just enough information for safe optimization—not just APIs, but the exact theoretical properties (consistency guarantees, conflict semantics, object-metadata mappings)—without over-abstracting or leaking unnecessary implementation details. For instance, in RDMASTM, object-level caching can safely skip redundant validations because the layering exposes both cache status and orec mappings (see Chapter 5). This deliberate interconnectedness—where layers expose both interface contracts and theoretical guarantees—is key to achieving strong performance without sacrificing composability.

Thesis: Concurrent programming has long accepted false dichotomies: programmability versus performance, generality versus efficiency, composition versus specialization—as fundamental tradeoffs. This dissertation demonstrates that these tradeoffs are not fundamental but rather artifacts of inadequate information exposure at layer boundaries. By identifying and exposing the right semantic invariants through stable, architectural contracts—orec values as proofs enabling operation snapshotting (exoTM), co-located but distinct ownership and versioning metadata avoiding false conflicts (Harmony), and object layout control, validation optimization API, and temporal locality hints enabling RDMA-aware optimization (RDMASTM)—systems can separate what must be correct (synchronization semantics) from what must be efficient (implementation strategies) while enabling safe, incremental optimization. Three systems validate this principle across shared memory, multi-structure composition, and distributed RDMA, demonstrating that right-sized architectural decoupling achieves both programmability and performance without rewriting data-structure logic.

We validate this thesis in three arcs: exoTM/STMCAS establishes the base layer. exoTM provides the minimal synchronization mechanism (orecs and clock), supporting two policies: STM for general programmability and STMCAS for hand-optimized operation-level synchronization. Harmony adds the transaction-management layer for composability. Operations continue to use STM/STMCAS for per-operation synchronization, while Harmony enforces transaction-level serializability and structured cleanup via co-located metadata. RDMASTM lifts the same field-level interface to an RDMA setting, demonstrating layer-by-layer optimization from bottom to top. At the lowest metadata/data access level, pluggable policies for orec placement, clocking, caching, and batching adapt to network costs and access patterns. At the

middle STM/synchronization level, optimizations such as trimming unnecessary orecs and cache-aware orec validation elision reduce synchronization overhead. At the highest data structure level, programmers exploit these exposed features for structure-specific optimizations—such as unordered-map array caching or skip-list key caching—to further reduce RDMA accesses. Throughout, data-structure logic remains largely unchanged.

The following sections detail how each arc demonstrates incremental optimization through layered separation.

1.4 Three Problem-Solution Arcs

1.4.1 Single Data Structure: exoTM/STMCAS

Implementers face a choice: use a general STM for programmability and portability, or hand-engineer fine-grained locking or non-blocking code for peak performance. exoTM/STMCAS [19] factors out a minimal mechanism (orecs and clock) and runs two policies over it: STM for a general path, and STMCAS for hand-optimized, operation-level synchronization. Concretely, exoTM provides an ownership-record-based two-phase locking (2PL) mechanism with optimistic reads that can be checkpointed and validated minimally, exposing just enough structure for policy layers to optimize common cases. These two policies coexist over the same mechanism, letting programmers start with STM and migrate hot paths to STMCAS without changing data-structure code or the mechanism's guarantees. Uncommon or tough-to-optimize paths use STM while common paths use STMCAS, achieving near hand-optimized performance without complicating the programmer's ability to reason about correctness. Across diverse workloads, STMCAS maintains com-

petitive performance while achieving peak performance reaching 1.35× the best alternatives (details in Chapter 3).

1.4.2 Composition Across Structures: Harmony

Transactions are attractive for providing all-or-nothing semantics when updating multiple data structures atomically. However, composing specialized data structures in one transaction can introduce "false conflicts" where structure-level metadata collides even though operations are semantically independent (e.g., updating a primary map and secondary index for the same key). Additionally, managing cleanup on commit or abort across multiple structures is challenging. Harmony [20, 21] adds the transaction-management layer: operations continue to use STM/STMCAS for per-operation synchronization, while Harmony enforces transaction-level serializability and structured cleanup via co-located but distinct metadata (operation metadata and transaction metadata residing together in each node, protected by the same synchronization). This avoids false conflicts among transactions while maintaining structure-aware cleanup at commit or abort. The result is a system of transactional data structures (TDSs) that supports range queries (uncommon in TDS systems) and achieves competitive or best-in-class performance on microbenchmarks and TPC-C, matching or exceeding prior systems.

1.4.3 Distributed Setting: RDMASTM

In RDMA, pointer chasing becomes network round-trips, hot metadata turns into network bottlenecks, and memory registration pins regions and restricts relocation, complicating dynamic allocation and pointer-rich structures. RDMASTM presents a layered architecture that separates application/data-structure logic, synchronization

semantics, and hardware access through an Access & Placement Substrate (APS). APS sits between data structures and the synchronization layer, exposing pluggable policies—object-level caching, validation optimization, batching, clock management, and placement control—while hiding hardware-specific details. The synchronization layer invokes APS to realize policies over the hardware fabric. Because the field-level API and transactional semantics remain stable, data-structure algorithm logic remains unchanged—programmers add structural optimizations like anchors for hot metadata or cache-aware layouts for spatial locality—despite fundamentally different RDMA performance characteristics. This separation enables systematic exploration of the RDMA design space, deriving general principles—explicit locality management, operation grouping, validation optimization—that apply beyond STM to lock- and CAS-based designs. Our exploratory evaluation on CloudLab deployments (1–8 nodes, four data structures, three workloads) reveals promising mechanisms: object-level caching with validation-skip optimization delivers the largest gains (nearly 2× throughput, 50–80% RDMA reduction) by collapsing pointer-chasing and skipping redundant orec checks for cached objects. Readset trimming provides orthogonal benefits (16–30% improvement) when data-structure semantics permit safe pruning. Other mechanisms show workload-dependent effects: clock variants differ by <20%; batching helps write-heavy workloads with concentrated memory (5–10%) but offers minimal benefit when distributed (within ±2–3%).

Across these arcs, we keep the northbound contract fixed and expose only the invariants higher layers rely on: (i) a deterministic object→ownership-record mapping (exoTM/STM/STMCAS), (ii) clear conflict rules (e.g., same ownership record with a write; Harmony adds cross-structure range/read vs. update), and

(iii) a version/clock discipline for snapshot reads and commit order (all three). This lets us swap or tune policies underneath—e.g., RDMASTM's object-level caching with cache-hit validation optimization—without changing data-structure code or semantics. This interdependence is deliberate: each layer's contract exposes just enough guarantee for the next to optimize safely. In exoTM/STM/STMCAS, policies rely on exclusive ownership and monotonic versions; Harmony relies on co-located per-node metadata and transaction-level conflict rules; RDMASTM reuses the same semantics and adds a cache-hit signal for validation elision.

1.5 Contributions

This dissertation introduces three research artifacts:

exoTM/STMCAS (Chapter 3): A shared-memory concurrency framework that separates a minimal clock and ownership substrate (exoTM) from the policies built upon it. The framework supports an STM path for programmability and an STMCAS path for hand-optimized composite operations, with both policies coexisting within the same program. This enables programmers to use simple transactional abstractions for uncommon paths while achieving near hand-optimized performance for common-case operations.

Harmony (Chapter 4): A transactional data structure system that separates per-operation synchronization from data-structure-level transaction conflict detection and cleanup through co-located but distinct metadata. The system introduces coroutine-based cleanup continuations that execute at commit or abort, ensuring structure-aware resource management without entangling synchronization logic. We provide three compatible techniques for expressing non-existence in sets and maps, and proof sketches establishing serializability of non-commuting concurrent

operations.

RDMASTM (Chapter 5): A layered RDMA-based framework that uses STM as an exploratory testbed to systematically study RDMA optimization opportunities through an Access & Placement Substrate (APS). APS sits between data structures and the synchronization layer, exposing pluggable policies—object-level caching, validation optimization, batching, clock management, and placement control—while hiding hardware-specific details. Data structure algorithm logic remains unchanged; programmers add structural optimizations like anchors or cache-aware layouts. Through exploratory evaluation on CloudLab deployments (1–8 nodes, four data structures, three workloads), the work identifies promising mechanisms: object-level caching with validation-skip optimization delivers nearly 2× throughput and 50–80% RDMA reduction; readset trimming provides 16–30% improvement when semantics permit; clock and batching show workload-dependent effects. The work derives general principles—explicit locality management, operation grouping, validation optimization—that apply beyond STM to distributed concurrent data structure design.

1.6 Organization

The remainder of this dissertation is organized as follows:

Chapter 2 provides the technical foundations necessary for understanding the contributions. Section 2.1 covers concurrent programming fundamentals, including linearizability, memory consistency models, and the challenges of implementing concurrent data structures. Section 2.2 discusses atomic memory operations, from basic CAS to multi-word MCAS and their limitations. Section 2.3 examines transactional memory systems, their trade-offs, and the motivation for architectural

decoupling. Section 2.4 introduces Remote Direct Memory Access (RDMA) technology and explains why distributed settings introduce fundamental new challenges for concurrent data structures.

Chapters 3– 5 present the three main projects that validate the thesis. Chapter 3 introduces exoTM/STMCAS [19], demonstrating architectural decoupling for shared-memory concurrent data structures. Chapter 4 describes Harmony [20, 21], extending the principle to composable transactional data structures. Chapter 5 presents RDMASTM, showing that architectural decoupling applies to distributed RDMA-based systems.

Chapter 6 concludes with a summary of contributions, discussion of broader implications, and directions for future work.

Chapter 2

Background

This chapter provides the technical foundations necessary for understanding the contributions presented in this dissertation. We begin by examining the fundamentals of concurrent programming, including the role of concurrent data structures and the correctness conditions they must satisfy (Section 2.1). We then discuss atomic memory operations, from basic compare-and-swap to multi-word extensions, and explain how these primitives complicate programmability—reasoning about all possible interleavings, avoiding deadlock, and maintaining correctness as new operations are added (Section 2.2). Section 2.3 examines transactional memory systems, their promise and limitations, and explains how these observations motivate the layered transactional approach used in this dissertation, including transactional data structures (TDSs) that compose operations across multiple structures. Finally, Section 2.4 introduces Remote Direct Memory Access (RDMA) technology and explains why extending concurrent programming techniques to distributed settings introduces fundamental new challenges that require careful architectural consideration.

2.1 Concurrent Programming Fundamentals

2.1.1 Concurrent Data Structures and Correctness

Concurrent data structures are fundamental to building high-performance software on multicore systems. They provide familiar interfaces while encapsulating complex synchronization logic to ensure correctness when accessed by many threads simultaneously. Dijkstra's foundational work in the 1960s formalized the role of synchronization [1, 2, 3], which coordinates multiple threads' access to shared resources to prevent race conditions—scenarios where the outcome depends on the relative timing of interleaved operations by different threads. Classic examples span both lock-free and lock-based designs, including the Treiber stack [22], Michael–Scott queue [23], and lock-based B-/B+-tree variants from the database literature [24], alongside optimistic structures such as Harris's list [4], Bronson's tree [7], and skip lists [6]. These illustrate the range of synchronization styles used to achieve linearizable methods and handle synchronization intricacies internally, shielding programmers from low-level challenges while maintaining correctness guarantees.

2.1.2 System Model

We consider a program with multiple threads executing concurrently on a shared-memory multicore system. Each thread has private memory (e.g., its stack) and can access a shared heap. The shared heap contains objects with well-defined methods; threads interact by invoking these methods, which perform reads and writes to shared memory.

This creates dataflow among threads: one thread writes to an object via a method, another thread reads that object via its own method invocation. The

relative timing of these method invocations creates a *happens-before* relation [25] that partially orders operations. The challenge of concurrent data structure design is to implement objects and methods that ensure correctness despite arbitrary interleavings of concurrent method invocations.

In this setting, concurrent data structures impose structure on the shared heap—it is not merely a flat array of atomic registers, but organized regions with high-level semantics (stacks, queues, trees, hash tables). The science of concurrent data structures is about designing these objects and their methods to be both correct and efficient.

2.1.3 Linearizability

Linearizability [26] is the standard correctness condition for concurrent data structures. It ensures that each method invocation appears to take effect atomically at some instant—called its *linearization point*—between the method's invocation and response. Furthermore, the overall system behavior must be equivalent to some sequential execution that respects the real-time order of non-overlapping operations.

Formally, a history is linearizable if:

1. Each operation appears to occur instantaneously at its linearization point
2. There exists a total order over all operations consistent with real-time order
3. This total order is a valid sequential execution of the data structure's specification

Linearizability simplifies reasoning: programmers can think about each method as if it executes atomically, without worrying about interleaving with other opera-

tions. This compositionality—linearizable components remain linearizable when combined—makes it the gold standard for concurrent correctness. It enables modular verification: prove each method correct in isolation, and the entire data structure is correct.

2.1.4 Memory Consistency Models

Language-level memory consistency models play an important role in the realm of concurrent programming [27, 28, 29]. These models define the behavior of memory operations in multi-threaded contexts and dictate how changes to memory by one thread become visible to other threads. Different programming languages and systems implement various memory models, ranging from sequential consistency [30], where memory operations appear to execute in the exact order specified by the program, to more relaxed models that allow for optimizations at the cost of more complex reasoning about memory operations [27].

Understanding these models is essential for developers, as it influences the design and usage of synchronization primitives and concurrent data structures. When a programming language provides a high-level memory consistency model, a correctly synchronized program will behave as intended across different platforms and environments. These language-level memory consistency models are defined in terms of language constructs, primarily locks and `atomic` variables. Sequential consistency constrains visibility of loads/stores; linearizability constrains method-level behavior of objects. SC does not by itself guarantee object linearizability. These constructs are not easy to use, so again it makes sense to encapsulate concurrent interactions in concurrent data structures.

These constructs are not easy to use correctly. Sequential consistency (the

language or hardware memory model) guarantees program-order visibility but does not ensure linearizability of high-level operations—programmers must still reason about which interleavings are safe. Locks handle multi-location updates but introduce deadlock risks when acquiring multiple locks; getting a global lock ordering correct across all methods is error-prone and brittle as code evolves. For data structures that require multi-location updates, this complexity is difficult to avoid when working directly with low-level primitives.

2.1.5 Implementation Challenges

While concurrent data structures provide high-level interfaces for ease of use, their development involves manipulation of low-level synchronization primitives that is hard to verify and maintain. Implementing concurrent data structures directly using hardware-level synchronization primitives such as atomic instructions and memory barriers is usually impractical at scale from a maintainability and verification standpoint. The correct usage of these low-level primitives is not always intuitive and can be contrary to higher-level programming practices. In particular, programmers must carefully analyze all potential interleavings among instructions performed by concurrent threads, and add appropriate synchronization instructions to prevent any interleavings that could lead to errors. This leads to complicated, hard-to-maintain code; even expert programmers struggle to use these primitive operations correctly and avoid subtle concurrency bugs. Additionally, adding new methods to a concurrent data structure exacerbates complexity: each new method's reads and writes must interact safely with every existing method. The reasoning burden grows super-linearly with the number of methods and the shared state they manipulate; without higher-level structure, ensuring cross-method correctness

becomes the bottleneck to evolution, and subtle races may only manifest under rare interleavings.

2.1.6 Operating System Primitives

Concurrency has long been the domain of operating systems, and modern operating systems provide many synchronization facilities to programs. Indeed, programming languages often provide lightweight wrappers around these, in order to achieve platform independence. The most significant of these are mutexes, semaphores, and condition variables. These do not merely synchronize: they also allow the operating system to de-schedule a thread when its progress is impeded by another thread. For example, when one thread has acquired a certain mutex, others wishing to acquire it will be de-scheduled, letting the operating system reassign blocked threads' cores to other threads who might be able to perform meaningful work.

The misuse of low-level synchronization primitives can precipitate several critical issues in concurrent programming; these are often exacerbated by the blocking behavior within the operating system. A deadlock occurs when two or more threads are unable to proceed because each is waiting for the other to release a resource, creating a standstill. Resource contention happens when multiple threads compete for the same resource, leading to performance degradation. In addition, failing to properly sequence the acquisition of locks can lead to starvation, a situation in which a thread is perpetually denied access to resources it needs to make progress. More details of these issues are covered in "The Art of Multiprocessor Programming" [8].

These costs motivate alternatives to blocking synchronization. One approach is *optimistic reads* with version checks (seqlocks [31]): readers check that a sequence number is even at start and unchanged at end; otherwise they retry. Another

technique is *commit-time (lazy) locking*, in which lock acquisition is postponed to commit time, keeping contention windows short. Both ideas appear in our transactional designs.

OS synchronization primitives couple to a local kernel and do not extend across VMs or machines; clusters cannot use an OS mutex. In distributed settings, spinning on remote memory generates continuous network traffic and is prohibitively expensive. Our designs avoid remote spinning and rely on policies that minimize remote polling, a theme developed in the RDMA background and chapter.

2.2 Atomic Memory Operations

2.2.1 Hardware Support for Atomicity

Modern hardware provides many instructions that are able to interact with memory in an atomic fashion. That is, they appear to happen "all at once", in a single, indivisible operation that does not interleave with concurrent instructions. The most obvious examples are loads and stores of sizes up to a machine word. Such loads and stores to the same location appear to be totally ordered: a load will never observe "part of" a store to the same location. In addition, swap operations are able to place a new value in memory and return the value that was previously there. Many modern processors also provide atomic versions of basic arithmetic and logic, such as atomic AND, OR, ADD, and SUBTRACT. These can update a memory location by reading, computing a new value, and placing that value back to memory, all in an atomic manner. Concretely, we encode synchronization metadata as 64-bit, 8-byte–aligned words so CPUs and RNICs can update them atomically. Orecs pack a lock/owner bit plus a 63-bit version; the global clock is a

64-bit counter advanced via FAA/CAS. `Field<T>` validates with one atomic orec load on first access; writes acquire/release with one CAS and stamp the commit version. Immutable payload fields (e.g., keys) are const; when mutation would widen critical sections, we use copy-on-write (allocate a new node and publish by pointer swap). These layouts avoid partial reads/fences and enable safe use of RNIC 8-byte atomics.

2.2.2 Compare-And-Swap and Multi-Word Extensions

Herlihy showed that the most powerful single atomic instruction is one that can conditionally perform its update [32]. The Compare-And-Swap (CAS) operation is a fundamental atomic technique in concurrent programming, allowing a value at a memory location to be changed only if it matches an expected value. Architectures also provide Load-Linked/Store-Conditional (LL/SC) pairs [33] with similar power to CAS. LL/SC avoids some ABA issues but may spuriously fail; in practice, it is often used in loops similarly to CAS. This ensures the operation's integrity in multi-threaded contexts. Extending this concept, Multi-word Compare and Swap (MCAS) [34, 35, 36] allows atomic operations on multiple memory locations simultaneously, providing a powerful tool for complex synchronization tasks that involve coordinating changes across several points in memory. This enhancement of CAS to MCAS marks a significant advancement in the handling of intricate concurrent operations. However, modern processors do not provide an MCAS instruction, it can be simulated in software via a handful of CAS operations. Software MCAS implementations build multi-word atomicity using CAS operations on shared descriptors. This raises practical challenges: descriptor placement (in-object vs. separate), descriptor reuse and ABA issues, helping protocols for

lock-freedom, and safe reclamation of both descriptor and object memory. These concerns tightly couple MCAS to memory management and the language memory model, limiting its ease of use.

2.2.3 Why MCAS Is Not Enough

MCAS alone does not make concurrent programming "easy" (even double-CAS is not a silver bullet [37]). Early implementations of MCAS did not provide support for read-only traversals of data structures, which are necessary for computing operands for MCAS. Furthermore, integrating MCAS into the rules of a programming language's memory consistency model is not trivial. Some recent works have addressed this issue, making the operation more programmable and efficient [38, 39, 15, 40]. However, MCAS is still a low-level primitive, operating on raw memory locations. The designer of a data structure that uses MCAS must manually reason about the memory locations involved. In addition, it is not straightforward to compose operations, e.g., by chaining several MCAS operations together into a large code block that appears to execute atomically. The programmer may also need to manually tune aspects like memory layout to optimize MCAS performance. Higher-level abstractions like transactional memory often introduce additional metadata, such as *ownership records* (orecs)—per-object or per-stripe locks that track which transaction currently owns write access to a region of memory—to coordinate concurrent access and detect conflicts. Crucially, TM systems also address memory management holistically: safe reclamation of objects, descriptor lifecycle management, and integration with language-level features like garbage collection. This broader scope makes TM more than just a synchronization primitive—it is a programming model.

2.3 Transactional Memory

2.3.1 The Promise of Transactional Abstractions

The challenges described above highlight the need for a mechanism that not only enhances programmability but also simplifies verification and extension of concurrent data structures. A desirable solution is an abstraction that allows for local reasoning—where each method can be treated as if it were a single atomic block. Conceptually, a coarse-grained lock over all shared data would provide such semantics but at the cost of eliminating concurrency; the goal is to retain the semantics while permitting scalable implementations. This approach would ensure that the methods remain correct even when new methods are added later. Such an abstraction would facilitate easier management and expansion of complex concurrent data structures, making them more accessible and reliable for developers.

Transactional Memory (TM) offers such a programming model. Programmers annotate lexically scoped regions of code as transactions; a runtime system—implemented in hardware (HTM) or software (STM)—ensures that transactions issued concurrently by different threads do not interleave in ways that violate a chosen correctness condition (typically opacity [41], or serializability). Opacity ensures that even aborted transactions only observe consistent states—strengthening serializability by preventing transient inconsistencies from being visible; we use opacity by default in this dissertation. TM was first proposed in the context of hardware, known as Hardware Transactional Memory (HTM) [9]. Subsequently, the concept was extended to software-only systems, leading to the development of Software Transactional Memory (STM) [10].

Transactions simplify the programming model by abstracting away the com-

plexity of thinking about mutexes or CAS instructions. An added benefit is that transactions can be composed into larger transactions. Composability allows transactional functions to be nested or called within other transactions, reducing the programmer's need to manually manage synchronization and significantly lowering the risk of deadlocks. Most STMs provide flattening semantics for nesting (a nested transaction merges with its parent); some systems support closed nesting, but it is not required for our results. Transactions manage synchronization internally: the TM system tracks read/write sets and detects conflicts, aborting and retrying as needed; implementations may use trylocks internally, but those details are hidden from the programmer. Harris's work [11] introduced language constructs (e.g., `retry` and `orElse`) that support condition synchronization and improve composability, helping programmers build complex concurrent operations without ad hoc lock protocols.

2.3.2 Ownership Records and Optimistic Concurrency Control

STM systems typically use *ownership records* (orecs)—metadata structures, either per-object or per-stripe, storing (i) an ownership/lock indicator and (ii) a monotonically increasing version [42, 43]. This dissertation follows TL2-style designs that use a global version clock: transactions acquire a start timestamp, and committers take a commit timestamp from the global clock to stamp their orecs.

Orec mapping strategies:

- **Per-object orecs**: Each object has its own dedicated orec, eliminating false sharing but increasing metadata overhead

- **Per-stripe orecs**: Objects hash to a shared table of orecs, reducing space but introducing false conflicts when unrelated objects map to the same orec

Optimistic execution protocol (TL2-style):

1. **Begin**: Read start_ts from the global version clock.
2. **Read**: On first access to an object, read its orec; if locked or version > start_ts, abort or retry. Read the data, then re-read the orec; if the orec changed or is locked, abort or retry. Otherwise, record the orec's version in the read-set.
3. **Write**: Buffer updates in the write-set; don't modify objects directly.
4. **Commit**: Acquire orecs for written objects (CAS on lock bits), advance the commit timestamp (fetch-and-add or CAS on the global clock), validate the read-set (check each orec is unlocked and its version matches the recorded value), apply buffered updates, stamp each orec with the commit timestamp, and release locks.

This orec-based approach separates synchronization metadata from object payloads, enabling concurrent readers when no writer holds the lock. Our dissertation builds on this foundation, exposing orec mappings and version disciplines as explicit layer contracts (Chapters 3, 4, 5).

2.3.3 Hardware Transactional Memory Limitations

Despite the advantages of transactional abstractions, it is important to recognize the challenges and limitations inherent in TM, particularly its hardware implementation (HTM). In brief, HTM tracks a transaction's read and write sets in private caches;

the cache-coherence protocol detects conflicts between concurrent transactions and forces at least one to abort and retry. This hardware path provides low-latency commits when conflicts are rare, because it avoids STM's software logging and validation overheads. However, there are significant challenges associated with HTM. These include well-known limitations: capacity bounds (transactions are limited by private cache sizes, so touching too much data or being preempted can cause aborts); conflict granularity at cache-line size, which induces false conflicts when unrelated fields share a line; and interactions with system features (e.g., system calls, I/O, interrupts) that can trigger aborts. Additionally, Intel's TSX Asynchronous Abort vulnerability (TAA [44]) led to microcode mitigations that effectively disable TSX on many processors, and has contributed to vendors opting not to implement HTM in some architectures. These challenges mean that even if side-channel vulnerabilities were addressed, other significant hurdles in HTM implementation would remain.

2.3.4 Software Transactional Memory Trade-offs

STM offers a general programming model, particularly when compiler instrumentation (e.g., the C++ TM Technical Specification) inserts read/write barriers and supports irrevocability (via an "irrevocable" mode or I/O-aware mechanisms) so that a transaction performing I/O can be forced to commit alone; in that setting, it prioritizes ease of use and generality. However, the dominant costs in STM are per-access instrumentation, logging (undo/redo), and validation of read sets, which can limit efficiency for highly optimized data structures. Software TM introduces unavoidable overheads from per-access instrumentation, metadata management, and read/write-set tracking—even under low contention. These costs

remain fundamental regardless of optimization strategy. Moreover, the common design choice of using a shared global version/sequence clock simplifies validation but concentrates synchronization pressure on a single counter, creating a bottleneck for workloads with frequent writers. This tradeoff between reduced instrumentation and increased contention on the global clock has been analyzed in prior work on hybrid and software TM systems [45], which shows how global clocking shifts the balance between concurrency progress and instrumentation cost. Conflicts cause abort/retry in all TM variants; our focus here is on instrumentation and validation costs rather than singling out aborts as STM-specific. However, performance can be significantly improved by optimizing STM for specific data structures or access patterns [13], tailoring STM to specific applications according to their underlying data organization [14], or allowing programmers to directly manage transaction metadata and control validation and execution [15]. Sacrificing some generality or ease of use in STM—for example, bypassing compiler-assisted paths (TM TS) or constraining supported features—can deliver substantially higher performance. Compiler-assisted STM provides generality (language integration, support for irrevocability) and ease of use (automatic instrumentation), but those benefits carry overheads that specialized designs can avoid.

2.3.5 Layered Decoupling and Stable Contracts

These trade-offs between generality, ease of use, and performance have led researchers to explore alternative designs that decouple low-level synchronization from language integration and high-level use *semantics*. This decoupling principle, well-established in operating systems design [16, 17, 18], makes it possible to give programmers fine-grained control over synchronization overhead while ensuring

mechanisms are used correctly. By *stable*, we mean the guarantees exposed upward do not change as implementations change below. Example: "operations conflict if they access the same orec and at least one is a write" remains true whether orecs are per-object or striped, in shared memory or over RDMA. Stability enables swapping implementations without rewriting data-structure code. By separating what operations *mean* (their guarantees and conflict rules) from how they are implemented (e.g., conflict detection, validation, caching policies), systems can adapt to diverse use cases without rewriting application logic. While this approach has been underexplored in the context of concurrent data structures and transactional systems, it offers potential benefits for shared-memory concurrency. Furthermore, extending this *decoupling* to distributed environments grants the programmer and runtime greater awareness of when remote memory accesses occur. Given their high cost, being able to precisely track—and avoid—such accesses is critical to high performance.

2.4 Remote Direct Memory Access

The preceding sections have focused on shared-memory concurrency, where all memory accesses are managed through a hardware-supported memory hierarchy (caches, DRAM, NUMA interconnects). However, modern distributed systems often require coordination across multiple machines connected by a network. Remote Direct Memory Access (RDMA) offers a way to build low-latency distributed data structures by allowing network interfaces to directly read or write application memory without kernel mediation on either host. However, it also introduces constraints that make a straightforward port of shared-memory techniques to RDMA impractical [46].

2.4.1 RDMA Fundamentals

RDMA exposes two families of "verbs" for interacting with remote machines: *memory-semantic* (one-sided) operations that directly access remote memory, and *channel-semantic* (two-sided) operations that exchange messages between endpoints. Applications interact with RDMA devices by allocating resources in a *Protection Domain* (PD), creating *Queue Pairs* (QPs) with send/receive queues, registering application buffers as *Memory Regions* (MRs) that are pinned and assigned local/remote keys (*lkey*/*rkey*), and polling *Completion Queues* (CQs) for work completions [46].

Transport capabilities matter. RC (Reliable Connected) supports full RDMA semantics—including SEND/RECV, RDMA READ/WRITE, and 64-bit atomics (CAS/FAA)—making it the most general and widely used transport. UC (Unreliable Connected) supports SEND/RECV and RDMA WRITE but does not provide RDMA READ or atomic operations, reducing both features and reliability. UD (Unreliable Datagram) supports only SEND/RECV and is typically used for control-plane or multicast-style communication. Hardware atomics operate only on 8-byte, naturally aligned locations [47, 48, 49].

RDMA WRITE with immediate extends a normal WRITE by delivering a small 32-bit immediate value that generates a CQE at the receiver (requiring a posted RECV). This mechanism is commonly used for lightweight notifications or control-plane signaling [47].

Because MRs must be registered before remote access, dynamic allocation typically relies on pre-registered pools and, ideally, huge pages to amortize registration cost. Modern adapters support *On-Demand Paging* (ODP) to relax pinning and registration requirements, but ODP trades simplicity for potential page-fault latency

and device limits, so many systems still pre-register hot data structures [47, 46].

How this matters to RDMASTM. Our framework uses preregistered memory regions with stable remote addresses, RC queue pairs for one-sided READ/WRITE and 8-byte atomics (CAS/FAA), and a thin substrate (Remus) that exposes these regions as addressable segments. To amortize per-verb costs we employ batching (using scatter-gather lists and selective signaling) and inline small writes where supported; we do not use UD transports or two-sided messaging in our evaluation. NIC atomics operate on 64-bit, 8-byte-aligned values, which shapes the orec and global clock encodings used by the transactional layer. [47, 46]

2.4.2 The RDMA Cost Model

To illustrate the key cost difference between shared memory and RDMA, consider a red–black tree lookup. In shared memory, reading a node's key and its child pointer often hits in cache due to spatial locality; walking a few levels incurs only a few cache misses. In RDMA, each node access (i.e., pointer dereference) typically requires a one-sided RDMA READ with its own network round trip. With careful layout you can fetch multiple fields with a single READ, but you still pay roughly one RTT per node hop. For small transfers, latency is dominated by the number of round trips, not the total bytes moved. Unlike shared-memory loads, one-sided RDMA READs do not benefit from local hardware-managed caching of remote memory—each READ goes over the network.

Terminology. We distinguish:

- **Batching**: grouping multiple RDMA verbs so that they share a single doorbell write (the host-to-NIC notification that posts work requests) and

reduce completion-queue processing (CQE generation and polling), thereby amortizing per-operation NIC overheads.

- **Pipelining**: issuing multiple outstanding verbs so the RNIC overlaps network latency with computation (completion pacing via CQ moderation)

Batching reduces MMIO/CQ overhead but does not reduce the number of network round trips; pipelining hides those round trips by keeping many RDMA operations in flight. [47, 46]

STM adds remote validations at commit (see Section 2.3.2), further increasing round-trip counts unless operations are coalesced into a single remote access or served from a transactional cache. When layering STM in a distributed setting, validation amplifies this effect: reading and later re-checking ownership records (orecs) adds more round trips unless we coalesce or cache. Fetching an entire object in a single RDMA operation and reusing it for subsequent field accesses avoids repeated network reads and orec validations. Combined with *cache-hit validation optimization* (skipping redundant orec checks for already-cached objects), cuts round-trip count dramatically. Chapter 5 demonstrates this approach.

Per-operation overhead. Each RDMA verb requires the host to post a work request, the network interface card (NIC) to process it, and for completions to be polled. Even though this bypasses the kernel, a network round trip remains microseconds, not nanoseconds. Qualitatively, on-core cache hits and DRAM are nanosecond-scale, whereas NIC round trips are microseconds (e.g., L1≈1 ns, L2≈4 ns, DRAM≈100 ns, RDMA RTT≈2–6 μs [50]); we defer detailed measurements to the RDMA chapter. State-of-the-art software stacks on commodity NICs report 2–6 µs end-to-end RPC latencies in the common case [51]. It is therefore crucial to batch operations (*scatter-gather lists*/doorbell batching) and coalesce

bytes/fields into fewer verbs to amortize fixed per-verb costs; for small payloads (e.g., 1–128B) latency is largely insensitive to size, so reducing verb count matters most [50].

Pointer chasing becomes expensive. In shared memory, a pointer dereference can be a cache hit ; with RDMA it results in a network round trip. Traversing a linked list or tree with n nodes may require n sequential RDMA operations. Successful RDMA systems avoid per-node round trips by laying out data contiguously (arrays, slabs) and issuing range reads, or by redesigning protocols to use one RDMA verb per high-level operation (e.g., FaRM's lock-free read) [52].

Hot metadata creates bottlenecks. Unlike CPUs with deep caches and coherent interconnects, RNICs have limited internal resources. Concentrating updates to hot locations (e.g., a global version counter) can saturate the target's NIC or PCIe links and interfere with other verbs; complex verbs (atomics) are especially costly [53, 50].

Memory registration constraints. Because remote one-sided access requires a valid remote key (rkey)—an access token returned when registering a Memory Region (MR) that authorizes remote QPs to access a specific address range, and that must be supplied with each READ/WRITE/atomic—the MR must remain registered for the key to be valid [47, 46]; allocators must coordinate registration and reclamation; unsafe reuse or deregistration while peers still hold rkeys leads to access errors. On-Demand Paging (ODP) mitigates the need for pre-registering large memory regions and allows lazy pinning, but it introduces different failure and latency modes (e.g., RNIC-triggered page faults and increased tail latency) [54, 55].

Ordering and visibility. RC provides in-order execution *per QP*, but there is no cross-QP ordering. When subsequent operations depend on the visibility of a

prior WRITE (e.g., a remote CPU polling a flag or a following READ), software must rely on CQ completions and, when needed, a fence—the verbs interface exposes `IBV_SEND_FENCE` for RC QPs only—to enforce ordering constraints [47].

2.4.3 Why Shared-Memory Techniques Do Not Translate

The characteristics above mean that high-performance shared-memory algorithms rarely perform well when naively ported to RDMA: algorithms designed for hardware-managed caching and low-latency local atomics do not translate directly to a setting where each remote access is a network round trip.

- **Validation multiplies bandwidth costs.** Optimistic validation of an n-item read set implies up to n remote reads, which can dominate run time—even if batched [50].

- **Fine-grained synchronization becomes impractical.** Lock-free structures that require multiple CAS operations on remote nodes suffer from per-atomic latency and the limited throughput of RNIC atomics; furthermore, only 64-bit atomics are available, and only on RC [47, 49].

- **Access patterns matter more.** Layouts that scatter related data cause excessive network traffic and many serial round trips, whereas contiguous layouts and bulk transfers (or replacing one-sided access with efficient RPCs) dramatically reduce remote operations; moreover, avoid *remote spinning*, which generates persistent NIC traffic and exacerbates congestion—ALock demonstrates how an asymmetric lock can synchronize local and remote accesses without loopback or RPCs while limiting remote spinning [52, 51, 56].

2.4.4 Implications for Distributed Data Structure Design

The challenges outlined above—validation bandwidth costs, expensive remote atomics, critical ordering requirements, and layout sensitivity—create tension between two common design approaches. Hard-coding aggressive optimizations (batched operations, carefully tuned caching, specialized memory layouts) can achieve good performance for specific workloads but resists adaptation when access patterns or consistency requirements change. Conversely, building general abstractions that completely hide RDMA details sacrifices the fine-grained control needed to manage these performance-critical trade-offs.

As with STM in shared memory (Section 2.3), a *decoupled, layered design* offers a path forward for RDMA. Rather than forcing a choice between performance and generality, an RDMA-based system can provide *stable abstractions and contracts*—transactional APIs, clear correctness guarantees, explicit object–metadata mappings—while exposing well-defined controls that let programmers tune performance without rewriting data structures. Crucially, *exposing semantics in the API* enables safe optimization; for example, orec-based conflict detection and version-stamped commits let us bulk read object state and validate once, which our cache-hit validation optimization exploits to avoid redundant checks within a transaction.

Consider caching as an example: exposing cache state (hit/miss) to the synchronization layer enables optimizations like skipping redundant validations when an object is already cached, while the abstraction ensures consistency guarantees remain intact. Similarly, batching strategies (scatter-gather lists, multi-key reads/writes) can group remote operations to amortize per-request costs [50], with configuration controlling batch sizes and flush points while the abstraction preserves

correctness guarantees. Metadata placement strategies can distribute hot data (like version clocks or ownership records) across nodes or cache it locally [53], responding to observed access patterns without altering the fundamental synchronization mechanism. Finally, synchronization granularity can adapt to RDMA costs—using coarser-grained critical sections or RPC-based designs where remote atomics would dominate [51, 50]—with the mechanism providing correctness guarantees regardless of granularity choices.

This architectural approach informs the design of the RDMASTM framework presented in Chapter 5. RDMASTM uses STM as an exploratory testbed to systematically study RDMA optimization opportunities through a layered architecture with an Access & Placement Substrate (APS). APS sits between data structures and the synchronization layer, exposing pluggable policies—object-level caching, validation optimization, batching, clock management, and placement control—while hiding hardware-specific details. The synchronization layer invokes APS to realize policies over the hardware fabric. Data structure algorithm logic remains unchanged; programmers add structural optimizations like anchors for hot metadata or cache-aware layouts for spatial locality. This exploratory approach yields general principles—explicit locality management, operation grouping, validation optimization—that apply beyond STM to lock- and CAS-based designs. Rather than claiming to have found optimal solutions, this approach enables systematic investigation of the RDMA design space, identifying promising mechanisms and workload-dependent effects while preserving correctness through stable contracts.

Chapter 3

exoTM and STMCAS: Separating Mechanism from Policy in Shared Memory

This chapter establishes the foundation for architectural decoupling in shared-memory concurrent data structures. When designing concurrent data structures (CDSs), it can feel like programmers must choose between performance and convenience. On one hand, Software Transactional Memory (STM) is easy, because it allows programmers to simply mark regions of sequential code as requiring atomicity, and the compiler and STM runtime ensure that all conflicting accesses are mediated through the transactional mechanism—preventing inconsistent interleavings from becoming visible and preserving opacity. However, this can easily lead to false conflicts that hinder scalability. On the other hand, programmers can craft custom lock-based or non-blocking protocols for operating on the CDS. This approach increases scalability, but is hard to get right.

For lock-based optimistic data structures, this chapter identifies a compelling design point that effectively unifies the two approaches. The key idea is to employ the systems concept of "separation of policy and mechanism". This allows different operations to use the same transactional synchronization mechanism in different ways, on the same data structure, at the same time. That is, some operations can access the CDS via STM, while others use custom synchronization.

This chapter defines exoTM as a synchronization mechanism extracted from a popular STM. It then introduces a high-performance exoTM-based multi-word compare-and-swap policy (STMCAS) for creating CDSs. STMCAS can checkpoint and resume read-only operations, and can run concurrently with an exoTM-based STM policy. This allows programmers to create CDSs by starting with STM and then incrementally optimizing with STMCAS until the desired performance is reached. Evaluation demonstrates that the low-level mechanisms are fast: STMCAS-based data structures usually outperform lock-based and lock-free CDSs, at roughly the same level of effort. Furthermore, for complex CDSs (e.g., red/black trees), STMCAS and STM can be mixed to keep complicated operations (re-balancing) simple while using STMCAS to optimize the common case (read-only traversal).

3.1 Introduction

Transactional Memory (TM) was first proposed as a hardware (HTM) [9] and then software (STM) [10] technique for simplifying the creation of concurrent data structures (CDSs). Researchers subsequently discovered the appeal of transactional lock elision [57, 58]: Replacing a function's critical sections with transactions facilitates composition [11], as transactional functions can be called from other transactions without risking deadlock. TM thus morphed from a low-level synchronization

mechanism into a first-class programming abstraction.

To make TM work in production and without HTM support required new STM algorithms [59, 60, 42, 61, 62, 63], optimizations [64, 65, 66, 67], integration with locks [68] and conditional synchronization [11, 69], support for I/O [70], transactional consistency guarantees [41], transactional progress guarantees [71, 72, 73, 74], and integration with language-level memory consistency models [75, 76]. The C++ TM Technical Specification [77, 78] addresses most of these issues. However, it is complex to implement and difficult to use [79]. In systems without HTM acceleration, general-purpose language-level STM rarely performs well enough to be useful. When STM is willing to sacrifice some generality and ease of use, it can be much faster [80].

Another approach to simplifying CDS design is to give the programmer a multi-word compare and swap (MCAS) primitive [34, 35, 36]. Recent works have made MCAS more programmable, faster, and easier to integrate with language-level memory consistency models [39, 15, 40]. In these works, the underlying synchronization mechanisms resemble STM, but the API and programming abstraction do not.

The existence of these works suggests that the principle of separating policy and mechanism can be applied. Language-level transactions and MCAS are policies. Low-level synchronization techniques, like HTM, versioned ownership records [81, 42], byte locks [82], value logging [83, 63], and quiescence [76] are the building blocks of different mechanisms upon which either policy can be built. From a systems perspective, it is good to separate policy and mechanism, especially if the mechanism can *simultaneously* support multiple policies, each suited to a specific use case [16, 17, 18]. Achieving and exploiting such a separation for STM is the focus of this chapter.

The first contribution is to isolate the low-level synchronization mechanism of the TL2 [42] and TinySTM [61] STM algorithms into a component we call "exoTM" **(Section 3.3)**. As with library operating system research [84, 85], exoTM is lightweight, narrowly defined, and protects itself from the policies that use it.

The second contribution exploits how exoTM exposes functionality that STM policies have overlooked or underutilized. This chapter introduces a new policy, called Software Transactional Multi-word Compare and Swap (STMCAS), which is intended for CDS designers and avoids almost all of the overheads of STM **(Sections 3.4 and 3.5)**. Unlike STM policies, STMCAS does not emphasize ease or generality. While the methodology for using STMCAS is straightforward, it requires the programmer to be aware of details of the underlying exoTM. In return, STMCAS performance significantly exceeds STM, and often outperforms lock-based and lock-free designs. Two keys to this performance are STMCAS's support for (1) allowing programmers to stitch together a sequence of read-only and writing steps into a larger operation that appears to execute atomically; and (2) checkpoint and resume of optimistic traversals.

The third contribution is to build STM policies with exoTM **(Section 3.6)**, and then show how they can run concurrently with STMCAS **(Section 3.7)**. Programmers can design their data structures to use STM for uncommon and complex code paths, and STMCAS for the common case. This is showcased by introducing a SpecTM-like policy [80] and showing how using it concurrently with STMCAS can simplify the implementation of a scalable balanced tree and resizable hash table.

Evaluation compares the policies against STM and locking/lock-free data structures **(Section 3.9)**. STMCAS performance is never below 75% of the state of the

art, and its peak is as much as 1.35× the performance of the best CDSs.

3.2 Background: STM Mechanisms

Language-level STM requires programmers to annotate code regions that require atomicity. The compiler transforms these regions by instrumenting the region boundaries and the memory accesses within the region. The instrumentation coordinates transactions via the following synchronization metadata.

Ownership Records (orecs): Abstractly, an orec is a pair ⟨`ownership`, `version`⟩ that can be read or written atomically. `acquire` returns `true` if it atomically moves `ownership` from the `UNOWNED` state to some other state, and `false` otherwise. `release` resets `ownership` to `UNOWNED` and may increase `version`. Often, threads store their unique ids in the `ownership` field. An orec can be packed into a single word by using the high bit to indicate if the remaining bits are a version or thread id. Orecs mediate conflicts as follows. (1) A transaction cannot read or write a location if its orec is owned by another transaction. (2) A transaction must `acquire` an orec before performing a write. (3) A transaction reads location l by ensuring that l's orec is un-owned and constant while it accesses l.

The mapping from addresses to orecs is a policy decision. Object-based STM places orecs in object headers [64, 59], and an object's orec protects all of its fields. Most language-supported STMs are word-based: they hash addresses into a table of orecs. If the hash granularity is too small, multiple orecs may protect a single field of an object; If it is too large, false conflicts are possible [61, 86]. The mapping must be fast, since it may be performed on every read and write. A common mapping is to shift and mask an address, so that nearby locations map to the same orec.

STM algorithms that do not use orecs still rely on metadata for ownership

and versioning. TLRW [82] and LarkTM [67] use readers/writer locks in place of orecs; write acquisition is ownership and read acquisition keeps a location's version constant. NOrec-like algorithms [63, 83] use as few as one orec for acquisition, and then use the values of program data to detect conflicts. RingSTM [62] and InvalSTM [87] express ownership by publishing transactions' write sets.

The Clock Memory accesses by transactions must appear as if they all happened at once. Suppose a transaction reads location l_1, sees value v_1, and sees orec version o_1. If it then reads l_2 and sees value v_2 / orec version o_2, how can it tell if the read of l_2 is consistent with the read of l_1?

Most STMs provide opacity [41], which ensures the consistency of the entire read set upon each new access. Early STM algorithms did so through *incremental validation*: after reading l_n, but before using v_n, the orecs for $l_1 \ldots l_{n-1}$ are verified to still hold the values $o_1 \ldots o_{n-1}$ [88].

Dice et al. used a shared memory clock to decrease opacity overheads [42]. In the most straightforward version ("GV1"), transactions start by reading the clock. When committing, they advance the clock and use that value as the version for all orecs they release. When reading, transactions compare orec versions to their start time. If an orec's version is $\leq$ the transaction's start, then locations associated with that orec could not have been updated since the transaction began. Dice et al. presented several variations of this clock algorithm; Marathe and Moir applied it to non-blocking STM [89]; and Ruan et al. replaced the shared memory counter with a hardware clock [90]. Other STM consistency policies, such as snapshot isolation, may also employ clocks [91].

The Epoch Table When a transaction wishes to perform I/O or other operations that cannot be undone, it may transition to an *irrevocable* mode, where it is the

only transaction executing [92, 93]. To *privatize* a datum (i.e., transition it to a state where it can be accessed non-transactionally), it might wait after committing, to ensure that any speculative accesses to the datum complete [76]. Memory reclamation is a form of privatization, and thus a general-purpose STM must address privatization for *all* transactions.

Many STMs use an "epoch table" for these purposes. The table allows threads to declare when they are executing a transaction, to block other threads from starting transactions, and to check if any concurrent transactions started before some specific time. A straightforward implementation has each thread store its start time and an `inTransaction` flag in a shared collection. Since start times are monotonically increasing, transactions can `quiesce` by scanning the table and waiting for each `inTransaction` thread's value to exceed some appropriate value.

Are There Other General Mechanisms? STM implementations vary in how (and even if) they use the above mechanisms. In some, committing writers advance the clock early, to potentially avoid the need to re-check the validity of all reads [42]. On abort, some STMs that use undo logging must release orecs to new version numbers, instead of releasing orecs to their old versions. Some STMs use `setjmp` to retry transactions, while others use exceptions [59] or compiler support [94]. Some use compiler sandboxing instead of opacity [95]. Some abort on any inconsistency [42], while others try to validate and recover [61, 96, 63]. There is also significant variation in how reads and writes are logged by active transactions, and in how conflicts among transactions are resolved [71, 97]. We conclude that there are no other mechanisms that are sufficiently common to warrant consideration as part of a core mechanism for STM.

3.3 Isolating the STM Mechanism

We define exoTM as a small collection of mechanisms that enable STM and other synchronization policies. Listing 1 presents the exoTM API, which consists of orecs (lines 1– 2) and a clock (line 3). We exclude the epoch table since it seems specific to language-level STM requirements (irrevocability and privatization). CDS implementations typically achieve safe memory reclamation without privatization/quiescence [98, 99, 100, 101, 102, 103, 104, 105].

When beginning, an exoTM operation assigns itself a start time (`assignStartTime`). It can clear its start time (e.g., upon completing an operation) via `clearStartTime` or `releaseOrecs`. When the clock uses the x86 `rdtscp` instruction, a memory fence is required between reading the clock and using the 64-bit value it produces [106, 90] (line 7). The `readStartTime` accessor allows policies to use `_start` to implement epoch tables.

`checkStart` determines if orec `o`'s value is compatible with the current operation's `start_time`, or if `o` is already acquired by the operation. If not, it returns `EndOfTime`, which is larger than any value returned by the `_clock`, and distinguishable from the bit pattern of any pointer to an `ExoTMThread` object. It also returns a Boolean indicating if the orec is owned (by the caller or another thread). The sibling to `checkStart` is `checkVal`, which specifically compares the orec value to an argument.

`acquireStart` only acquires an orec whose version is consistent with `_start`. It returns a pair where `ok` indicates if the caller acquired the orec, and `locked` indicates if the orec was locked when the call was made. `acquireVal` mirrors the behavior of `checkVal`: it acquires an orec not if it is consistent with the caller's start time, but instead consistent with some provided value(e.g. orec value). `acquire`

ignores the caller's start time, and acquires the orec if it is not acquired by another thread. On every successful acquisition, the orec is saved to a per-thread log (`_owns`). Note that exoTM forbids threads from releasing each others' orecs.

`releaseOrecs` is responsible for releasing all of the orecs held by a thread. It can release in three ways:

- `NewTime`: Install new times in orecs, to indicate that values have been updated.
- `OldTime`: Restore prior times in orecs, to indicate that values did not change.
- `IncTime`: Restore prior times, but increment them, to indicate that a write was undone. This mimics incarnation numbers [61] in STM. If the clock is a shared counter, line 31 ensures the orec value does not exceed the clock.

We argue that the exoTM interface is *safe*, that is, a programmer cannot write code that puts the exoTM's objects into an invalid state., because it guarantees the following properties:

- Orec versions increase monotonically.
- Orec versions always reflect clock values.
- When an orec is owned, other threads cannot acquire it.
- Only exoTM can determine the values of the clock and orecs.

Note that safety does not imply that a programmer cannot write a racy program using exoTM: It is always possible to forget to interact with orecs, or use the wrong argument to `releaseOrecs`. Such errors relate to policies, not the exoTM mechanism.

Invariant #1: Threads cannot decrease the clock. Threads cannot decrease the global clock. exoTM requires only that the clock be globally monotonic and controlled by exoTM itself; it does not require that the clock be hardware-provided. On systems with hardware timestamp counters, `inc` and `now` simply read the clock to receive a value larger than any previous read. (If they observe an old value, they simply re-read.) However, even if a shared memory clock were used (where `inc` increments a counter), a malicious operation still could not reduce the clock value: exoTM reports values of the clock to the programmer, but never solicits a new clock value from the programmer.

Invariant #2: Threads cannot arbitrarily change the value of an orec through the exoTM API. There are only two ways to write to an orec. The acquisition methods use `CAS` to ensure that an orec will only be acquired from an un-acquired state: A thread can never acquire an orec that is owned by another thread. Furthermore, acquisition can only set the orec to values determined by exoTM. Lastly, orecs are only released via `releaseOrecs`, in which exoTM determines both which orecs to modify, and what values to write. Thus policies cannot forge versions for exoTM to write into orecs, or trick exoTM into releasing orecs that they did not acquire.

Note that safety does not imply that a programmer cannot write a racy program using exoTM: It is always possible to forget to interact with orecs, or use the wrong argument to `releaseOrecs`. Such errors relate to policies, not the exoTM mechanism.

Listing 1: exoTM API. "_" indicates private fields

```
class ExoTMThread // one per thread
    opaque type Orec // Packed into one machine word
1       _owned: boolean // 1 bit
2       union: { _version: uint63, _owner:ExoTMThread * } // 63 bits

3   singleton _clock: Clock // Represents time as 63 bits
4   const EndOfTime: uint63 // > Clock.max, ≠ ExoTMThread ptr

5   _owns: Set⟨Orec*, uint63⟩ // set of acquired orecs / old versions
6   _start: uint63 // Validity range for orecs; Initially EndOfTime

    function assignStartTime()
7       _start.atomicExchange(_clock.now()) ; return _start ;

8   function clearStartTime(): _start ← EndOfTime
9   functionreadStartTime(): return _start
    function checkStart(o: Orec *)
10      ⟨locked, ver⟩ ← *o
11      if ¬locked ∧ ver ≤ _start then return ⟨locked, ver⟩
12      if locked ∧ ver = this then return ⟨locked, ver : 0⟩
13      return ⟨locked, ver : EndOfTime⟩

    function checkVal(o: Orec *, exp: uint63)
14      ⟨locked, ver⟩ ← *o
15      if locked then return ⟨locked, ok : ver = this⟩
16      return ⟨locked, ok : ver ≤ exp⟩)

    function acquireStart(o: Orec *)
17      ⟨locked, ver⟩ ← *o
18      if locked ∧ ver = this then  return ⟨ok : true, locked : true⟩
19      if locked ∨ ver > _start then  return ⟨ok : false, locked⟩
20      if CAS(o, ⟨false, ver⟩, ⟨true, this⟩) then
21          _owns.insert(⟨o, ver⟩) ; return ⟨ok : true, locked : false⟩
22      return ⟨ok : false, locked : false⟩

    function releaseOrecs(how: enum)
23      end ← 0 ; clearStartTime()
24      if how = NewTime then
25          end ← _clock.inc()
26          for ⟨o, _⟩ in _owns do o ← ⟨false, end⟩
27      if how = OldTime then
28          for ⟨o, ver⟩ in _owns do o ← ⟨false, ver⟩
29      if how = IncTime then
30          for ⟨o, ver⟩ in _owns do o ← ⟨false, 1 + ver⟩
31          if max(o.ver, o ∈ _owns) = _clock.now() then _clock.inc()
32      _owns ← {} ; return end
    ---------------------------------------------------------------
    function acquireVal(o: Orec *, exp: uint63)
33      ⟨locked, ver⟩ ← *o
34      if locked then return ver = this
35      if ver ≠ exp then return false
36      if CAS(o, ⟨false, exp⟩, ⟨true, this⟩) then
37          _owns.insert(⟨o, ver⟩) ; return true
38      return false
    function acquire(o: Orec *)
39      ⟨locked, ver⟩ ← *o
40      if locked then return ver = this
41      if CAS(o, ⟨false, ver⟩, ⟨true, this⟩) then
42          _owns.insert(⟨o, ver⟩) ; return true
43      return false
```

3.4 The STMCAS Policy

The STMCAS synchronization policy combines desirable properties of STM and optimistic locking. Like STM, it enables programmers to create scalable, low-latency read-only steps that have an intuitive correctness criteria. From optimistic locking, it provides the ability to perform a targeted update of a small set of locations.

The key insight is that the programmer can use *orec values* as proofs of immutability. These proofs can be passed from one step of a data structure operation to the next, so that each step can establish its validity based only on the orec values that the programmer forwarded from the previous step. This fits nicely with a common CDS design pattern, where operations begin with a read-only traversal, after which there is a small writing step. Orecs provide an elegant way to transition safely from the first step to the second.

The performance of STMCAS comes from two sources. First, on account of the way STMCAS encourages programmers to break operations into steps, orecs can be used in an efficient manner without any downsides. Second, programmers can selectively optimize their use of orecs according to the semantics of the data structure.

3.4.1 The STMCAS API

Listing 2 presents the global and per-thread STMCAS fields and methods. The main feature is a mechanism for mapping orecs to program data (the `Ownable` type). STMCAS can either place orecs directly in `Ownables` ("Option 1") or maintain a global table of orecs ("Option 2"). In the latter case, the ownable-to-orec mapping is computed during `Ownable` construction, which reduces the overhead for complex hash functions that are more effective than a traditional shift-and-mod. `_getOrec`

Listing 2: The STMCAS Object

```
    class STMCASThread
        class Ownable
            constructor (stmcas: STMCASThread) // Option 1
1               initialize this._orec as Orec
            constructor (stmcas: STMCASThread) // Option 2
2               initialize this._orec as &_orecTbl[hash(this, NumOrecs)]
3           singleton _orecTbl: Orec [NumOrecs ] // Option 2

4       smr: SMRContext                         _exo: ExoTMThread
5       _start: uint63                          _lastRelease: uint63
6       snapshots: Vector ⟨Ownable, uint63⟩     rng: Prng

        function hash(a: any, upper: uint63)
7           return mix13Hash(a) mod upper
8       function _getOrec(o: Ownable): return & o._orec // Option 1
9       function _getOrec(o: Ownable): return o._orec // Option 2
10      function onStart(): _start ← _exo.assignStartTime()
11      function onRoEnd(): _exo.clearStartTime()
12      function onEnd(): _lastRelease ← _exo.releaseOrecs(NewTime)
13      function onFail(): _exo.releaseOrecs(OldTime)
        function checkStart(o: Ownable *)
14          return _exo.checkStart(_getOrec(o)).ver
        function acquireStart(o: Ownable *)
15          return _exo.acquireStart(_getOrec(o)).ok
        function checkVal(o: Ownable *, exp: uint63)
16          return _exo.checkVal(_getOrec(o), exp).ok
        function acquireVal(o: Ownable *, exp: uint63)
17          return _exo.acquireVal(_getOrec(o), exp)
18      function acquire(o: Ownable *): return _exo.acquire(_getOrec(o))
```

hides the differences between these two options. Since many data structures require hashing or random number generation, STMCAS also provides per-thread objects for these roles. It also provides each thread with a safe memory reclamation (SMR) algorithm, which we describe below.

For the most part, STMCAS is a thin wrapper over exoTM. It shadows the public methods of exoTM, and records the return values from `assignStartTime` and `releaseOrecs`. (Accessor methods for reading these values are omitted from the listing). The key difference is that the orec methods take a reference to an `Ownable`. This allows the programmer to define the mapping from data to orecs (which opens the door for more programmer-defined optimizations), without having

the ability to directly access the orecs (which protects the underlying exoTM mechanism). Typically, every node in a linked data structure will inherit from `Ownable`, to associate an orec with all the fields of that node. Alternatively, programmers can embed multiple `Ownables` into an object so different fields can be protected by different orecs, or explicitly map several objects to the same orec.

3.4.2 Fast Read-Only Traversal and MCAS Steps

To illustrate how STMCAS facilitates fast read-only operations, Listing 3 presents the traversal routine for an ordered map implemented as a doubly-linked list (dlist). Each dlist node descends from `Ownable`, and stores a single key/value pair, along with predecessor and successor pointers.

In the main traversal loop, the programmer reads all fields of the node pointed to by `next`, and then performs a single orec validation of the node (line 9). Note that it does not log orecs for re-validation. This manner of traversal was pioneered by TL2 [42]: If any orec's version is too new, the operation restarts; otherwise the entire set of reads is opaque [41] and consistent with the state of the data structure immediately before the call to `onStart`. This is a stronger consistency guarantee (multi-location atomic reads) than in most non-TM data structures, and is also easier to reason about. It is appealing that STMCAS can offer this with only a handful of additional instructions in each loop iteration. For completeness, we observe that since exoTM uses a hardware clock, orecs need only be checked *after* accessing fields, not *before and after* [90].

Listing 4 shows how the programmer can use orecs to craft efficient multi-location atomic updates. Consider a correctly synchronized program, where no shared datum is modified without first acquiring the associated orec. If STMCAS

Listing 3: Opaque read-only traversal with STMCAS

```
   type Node ⟨K, V⟩ extends Ownable
 1 |   key: K const                  val: atomic V
 2 |   next: atomic Node ⟨K, V⟩      prev: atomic Node ⟨K, V⟩
   type List ⟨K, V⟩
 3 |   head: Node ⟨⊥, _⟩   tail: Node ⟨⊤, _⟩   // Initially linked to each other
   function List::getLEQ(me: STMCASThread, key: K)
 4 |   me.onStart()
 5 |   curr ← ⟨n : head, v : 0⟩ ; next ← atomicRead(curr.n.next)
 6 |   if (curr.v ← me.checkStart(curr.n)) = EndOfTime then goto 4
 7 |   while true do
 8 |   |   nnext ← atomicRead(next.next) ; nkey ← next.key ; nver ← 0
 9 |   |   nver ← me.checkStart(next)
10 |   |   if nver = EndOfTime then goto 4
11 |   |   if nkey > key then me.onRoEnd() ; return curr
12 |   |   if nkey = key then me.onRoEnd() ; return ⟨next, nver⟩
13 |   |   curr.v ← nver ; curr.n ← next ; next ← nnext
```

Listing 4: List insertion via a follow-on step

```
   function List::insert(me: STMCASThread, key: K, val: V)
 1 |   me.snapshots ← {} ; me.smr.enter()
 2 |   ⟨node, ver⟩ ← getLEQ(me, key)
 3 |   if node.key = key then return false
 4 |   me.onStart()
 5 |   if ¬me.acquireVal(node, ver) then me.onRoEnd() ; goto 2
 6 |   next ← node.next
 7 |   if ¬me.acquire(next) then me.onFail() ; goto 2
 8 |   n ← new node (key, val, next, node)
 9 |   node.next ← n ; next.prev ← n
10 |   me.onEnd() ; me.smr.exit() ; return true
```

step S_1 observes value v in un-acquired orec o, and then step S_2 observes that o is not acquired and its value is still v, it can conclude that any data associated with o must not have changed in the interval between when S_1 accessed o and the present time. In our list example, this means a hypothetical second step could begin by using `checkVal` to ensure that the orec values returned by `getLEQ` have not changed. Symmetrically, `acquireVal` will acquire an orec only if the orec still holds the value observed by a preceding `check`, possibly from an earlier step. This is precisely the behavior of `insert`: the writing (MCAS) step can continue where the previous read-only step left off, by conditionally acquiring the returned orecs at their returned values. If it cannot, then it retries the `getLEQ`.

3.4.3 Programming with STMCAS

In Listings 3 and 4, we saw that STMCAS requires the programmer to explicitly interact with orecs. This is mostly mechanical, but represents the first of three ways in which programming with STMCAS can be considered "more difficult" than programming with STM.

Second, STMCAS requires the programmer to manage the mapping between program data and orecs. In this regard, it is similar to fine-grained locking, albeit without as much need to worry about lock order and deadlock. In more detail, if any thread T wishes to modify location l, all threads of the program must agree about the identify of the orec associated with l, and T must acquire that orec before any modification of l if it is possible that any other thread might simultaneously access l.

Lastly, unlike STM STMCAS does not interact with program data. This forces the programmer to think carefully about what data is immutable, with the remainder typically needing to be `atomic` variables. In C++, this includes both the data stored in the data structure, and also the pointers and other metadata [107]. While it may be possible for the programmer to produce their own redo or undo logging mechanism, we encourage a style of programming in which a writing step acquires all orecs before performing any writes, and is thus immune to consistency violations (and hence the need to retry) before writing to program data.

3.4.4 Optimizing STMCAS Operations

Reducing the Cost of Inconsistency: In Listings 3 and 4, the failure of a call to `checkStart`, `acquireVal`, or `acquire` can cause the entire list traversal to restart. Often, this leads to a re-computation of some prefix of the transaction.

For example, suppose `getLEQ` traverses the first 10 elements in a list, then finds the 11th's orec is acquired. Its local state will be reset, and it will restart at the list head. Instead, a programmer could periodically record the addresses and orec values it observed , and store them in the `_snapshots` field of the `STMCASThread` object. Upon detecting an inconsistency, the programmer could use `checkVal` upon returning to Line 5, to check if some recent entry in `_snapshots` was still valid. If so, the programmer can resume from one of these recorded states.

Broadly, this technique applies to operations where the nth read is dependent on the consistency of some small set of k preceding reads. A read-only step can periodically record a set of k locations and their orec values. If it later discovers an inconsistency, it can check these k-sized sets, in reverse order, and resume from the most recent set that is still valid.

Consistency-Aware Readers STMCAS requires programmers to manually avoid unnecessary instrumentation. For example, when an orec protects many fields of object o, the programmer can read many of o's fields, then `checkStart` once. A limitation arises when accessing computed fields of o. For example, if o has a fixed-size array ($o.A$), and mutable field $o.F$ is used to determine an array index ($f(o.F)$), then the programmer may need to `checkStart` after reading $o.F$, to prevent an artificial index-out-of-bound error in $o.A$.

A more aggressive optimization is possible if writing steps acquire all orecs before writing any program data, and write each location exactly *once*: If the programmer can show that a concurrent reader can use new values that are inconsistent with its start time, then they may elide some `checkStart` calls. In this regard, a traversal of linked nodes can approximate the consistency obliviousness of the Lazy List [108], particularly in cases where only one mutable field of each node is accessed during

the traversal (e.g., a `next` pointer).

3.5 A DList with STMCAS

In Section 3.4, we outlined the design principles of STMCAS, and gave an overview of how to program a dlist with STMCAS. In this section, we discuss some of the finer points, particularly with regard to optimization. As in Section 3.4, we will use the example of a dlist.

Pseudocode for , `remove`, `get`, and `update` appears in Listing 5. The dlist types are unchanged from Listing 3. As appropriate, we will also discuss finer points of the `insert` operation from Listing 4.

The Steps of DList Operations: `get`, `update`, `remove`, and `insert` follow a pattern: They use `getLEQ` to find the node with the largest key $\leq$ the target key. They operate on that node by reading it, updating it, removing it, or stitching a new node between it and its successor. In the listing, `getLEQ` is one step (delineated by its own calls to `onStart` and `onRoEnd`), and the remainder of `get`, `update`, `remove`, or `insert` is a separate step. The sequence of steps for each operation is fully enclosed between `smr.enter` and `smr.exit`. This is essential: Since the entire operation is enclosed in an `smr` region, object references returned by `getLEQ` are guaranteed to remain valid in subsequent steps.

We begin by discussing `get` and `update`. When `getLEQ` returns on line 2 or line 8, it returns a node and time, with a guarantee that the node was valid as of that time. `smr` ensures that this node will not be reclaimed before the operation completes, and its `key` is constant. If `key` is not found, we can linearize the operation when `getLEQ` returns, without a second step (line 3 or 9). When `key` matches, `get` uses a second step to read the value and then re-check the orec

Listing 5: An Optimized Doubly-Linked List with STMCAS

```
    function List::get(me: STMCASThread, key: K)
1       me.snapshots ← {} ; me.smr.enter()
2       ⟨node, ver⟩ ← getLEQ(me, key)
3       if node.key ≠ key then return ⊥
4       me.onStart() ; val ← node.val
5       if ¬me.checkVal(node, ver) then me.onRoEnd() ; goto 2
6       me.onRoEnd() ; me.smr.exit() ; return val
    function List::update(me: STMCASThread, key: K, val: V)
7       me.snapshots ← {} ; me.smr.enter()
8       ⟨node, ver⟩ ← getLEQ(me, key)
9       if node.key ≠ key then return ⊥
10      me.onStart()
11      if ¬me.acquireVal(node, ver) then me.onRoEnd() ; goto 8
12      node.val ← val
13      me.onEnd() ; me.smr.exit() ; return true
    function List::remove(me: STMCASThread, key: K)
14      me.snapshots ← {} ; me.smr.enter()
15      ⟨node, ver⟩ ← getLEQ(me, key)
16      if node.key ≠ key then return false
17      me.onStart()
18      if ¬me.acquireVal(node, ver) then me.onRoEnd() ; goto 15
19      prev ← node.prev
20      if ¬me.acquire(prev) then me.onFail() ; goto 15
21      next ← node.next
22      if ¬me.acquire(next) then me.onFail() ; goto 15
23      prev.next ← next ; next.prev ← prev ; me.smr.reclaim(node)
24      me.onEnd() ; me.smr.exit() ; return true
    function List::getLEQ(me: STMCASThread, key: K)
25      me.onStart() ; curr ← ⟨n : head, v : 0⟩
26      if ¬me.snapshots.empty() then curr ← me.snapshots.top
27      next ← atomicRead(curr.n.next)
28      if curr.n = head then
29          curr.v ← me.checkStart(curr.n)
30          if curr.v = EndOfTime then goto 25
31      else if ¬me.checkVal(curr.n, curr.v) then
32          me.snapshots.popBack() ; goto 25
33      snapshotCount ← SnapThresh
34      while true do
35          nnext ← atomicRead(next.next) ; nkey ← next.key ; nver ← 0
36          if nkey > key then
37              curr.v ← me.checkStart(curr.n)
38              if curr.v = EndOfTime then goto 25
39              me.onRoEnd() ; return curr
40          if nkey = key then
41              nver ← me.checkStart(next)
42              if nver = EndOfTime then goto 25;
43              me.onRoEnd() ; return ⟨next, nver⟩
44          snapshotCount ← snapshotCount − 1
45          if snapshotCount = 0 then
46              curr.v ← me.checkStart(curr.n)
47              if curr.v = EndOfTime then goto 25
48              me.snapshots.pushBack(curr) ; snapshotCount ← SnapThresh
49          curr.n ← next ; next ← nnext
```

(line 5). If this fails, it could mean that the node was removed from the list, that its value was modified, or that its predecessor or successor changed. In all of these cases, we return to `getLEQ`, resuming from the last valid snapshot. Note that `update` is almost identical to `get`, differing only in that it uses `acquireVal` instead of `checkVal`, overwrites `val` instead of reading it, and concludes with `onEnd` instead of `onRoEnd`.

`insert` is similar to `get`; the point of interest is employing `acquireVal` and `acquire` to insert a node. As with `get`, we can conclude that if the return value of `getLEQ` is unchanged (indicated by `acquireVal` succeeding on line 5), then it is not possible for `key` to be in the list. Consequently, we can use `acquire` instead of `acquireVal` to acquire the successor.

There are three subtleties in `remove`. First, since it traverses a predecessor link, we must be sure there are no races with an `insert` that sets the link. This is guaranteed by the fact that `insert` acquires its new node's predecessor and successor before creating the node. Consequently, if `remove` has acquired its target, it cannot be acquired by an `insert`, and thus its `prev` field is safe to read. Second, we `acquireVal` the target, and then `acquire` its predecessor and successor, which avoids the need for a `deleted` bit, or any sort of marking of reclaimed nodes: the change to the orec will invalidate any concurrent `getLEQ` that returns the same node. Third, when `smr.reclaim` is called, `remove` is guaranteed to succeed, and can immediately pass the node to the `smr`.

Optimizing `getLEQ`: Listing 5 presents a version of `getLEQ` with two optimizations. First, it makes use of orec values as *snapshots*, so that an operation can avoid traversing the entire list if it fails in a `checkVal` or `acquireVal` and calls `getLEQ` again. Second, it leverages knowledge of the data structure to avoid checking some

orecs at all.

All operations clear `snapshots` before their first call to `getLEQ`. With frequency 1/`SnapThresh`, `getLEQ` records the current node and its orec value in `snapshots` (lines 44 to 48). If `getLEQ` encounters an inconsistency and retries, it tries to start from the most recently added snapshot (line 26). It checks if that snapshot is still valid (line 31). If not, it discards the snapshot and tries an older one (line 32). If the check succeeds, then the `while` loop resumes from that snapshot. Note that `smr` ensures that entries in the `snapshots` collection can be checked for the duration of the enclosing operation.

Another key to CDS performance is optimistic traversal without verifying or validating each node along the way [109, 108]. Since the correctness of list operations only depends on the nodes returned by `getLEQ`, operations do not need to validate after acquiring their orecs in `update`, `insert`, and `remove`. This avoids many of the false conflicts common to STM-based lists, and should result in lower latency and better scaling. In addition, our optimized `getLEQ` does not check orecs on each iteration of the loop (lines 34– 49). This avoids latency, ordering constraints, and the potential for unnecessary aborts.

`getLEQ` never returns a node without checking its orec. Observe that `remove` acquires the orec of its target, and `getLEQ`'s orec check before returning (lines 37, 41, and 46) ensures that a node is never returned if it was concurrently removed. This is essential to the linearization argument for operations that fail lines 3 and 16 of Listing 5, and line 3 of Listing 4. Next, note that `getLEQ` only reads one mutable field of each node, and therefore it need not check orecs to ensure an atomic read of the node state: A mutating operation only writes to `next` once, so without an orec check, the read of `next` is still guaranteed to return either the old consistent value

or the new, and in either case, it will be consistent with the immutable `key` field. When inserting a node, the node's next pointer is initialized before its predecessor is linked to it (line 8), and its predecessor is linked to it before its successor (line 9). Furthermore, we never clear a node's successor pointer when removing it. Taken together, we arrive at roughly the same correctness argument as used for the lazy list [108]: If an optimistic traversal in `getLEQ` misses its target due to inconsistency, it stops at a node whose orec is too new, which will cause it to restart. (Note that we still must check any orec before adding it to `snapshots`).

3.6 Building STMs from exoTM

We implemented a compiler-supported STM (xSTM) using exoTM. xSTM can be customized to use undo or redo logging, encounter or commit-time locking, and several common techniques for opacity. This allows it to mimic most orec-based STM configurations [61, 42, 90]. xSTM implements its epoch table as a list of transaction start times, determined by calls to exoTM's `assignStartTime`, and thus supports irrevocability, I/O, and quiescence. xSTM is compatible with TM compiler instrumentation [78, 110, 111].

Given exoTM, implementing xSTM is straightforward. Listing 6 presents a variant with undo logging, early orec acquisition, and two orec checks per read. It resembles the write-through version of TinySTM [61]. In the listing, we omit the internal mechanisms for contention management and nesting, since their implementations are orthogonal to exoTM mechanisms. Likewise, we omit the irrevocability code paths, since they only indirectly interact with exoTM (via the provided `quiesce` method).

As with traditional compiler-oriented STM algorithms, we map addresses to

entries in a table of orecs (_getOrec). This is analogous to "Option 2" in Listing 2, but since the mapping is called much more frequently (on every load and store), a simpler hash function is used. txBegin saves the register checkpoint and gets a start time. txCommit uses a fast path for transactions that do not perform writes, but still calls quiesce (line 33) before reclaiming any shared memory. When the transaction has pending writes, it validates (using checkVal) before releasing its orecs. If the call leads to an abort, it uses the IncTime argument to release orecs, so that concurrent calls to txRead do not experience out-of-thin-air reads. Our redo-based implementations use the OldTime argument to releaseOrecs when aborting; our version that uses undo but only checks orecs once per txRead must use NewTime when releasing orecs upon abort.

txRead employs checkStart to ensure that any location it reads is consistent with all previous reads. When the check fails, it uses the locked indicator to identify possible deadlocks; when it is false, txRead tries to extend its timestamp (lines 20– 21).

In this version of xSTM, txWrite updates memory via undo logging. It relies on exoTM to detect attempts to acquire an orec already held by the transaction, via acquireStart, and it also uses timestamp extension when an orec is encountered that is unlocked but "too new" (lines 29– 30).

We note one instance in which the behavior of this xSTM implementation differs from TinySTM. In prior work that used a logical clock (shared counter), txCommit could increment the clock before line 35, and skip validating if the clock advanced by exactly one relative to _start. In contrast, exoTM's releaseOrecs method advances the clock and releases orecs, meaning that it would not afford this optimization if its clock were implemented as a counter. This could be addressed

by adding a lambda parameter to `releaseOrecs`, which could validate conditioned on the result of line 25, before executing line 26. Since the exoTM implementation in this chapter uses hardware clocks, which do not afford this optimization, this feature is not shown in the pseudocode.

3.7 Using STM with STMCAS

In order for an STM policy to be able to run concurrently with STMCAS, there are three broad requirements. First, the policies must agree on the mapping from locations to orecs. Otherwise, they might use different orecs to try to synchronize access to the same locations, which could be racy. Second, the STM should not interact with *program data* in a manner that is incompatible with the optimizations that an STMCAS user may employ. Most notably, the "Consistency-Aware Readers" optimization in Section 3.4.3 outlines two requirements for an operation: it must acquire all orecs before writing any program data, and it must only update each location once. Third, the use of STM features must be limited to those that are part of the common exoTM mechanism.

To address the second requirement, the STM policy cannot use undo logging. It may still use either encounter-time locking or commit-time locking [112, 61], but speculative writes must be kept in a redo log. Furthermore, that redo log must not allow duplicate entries for the same address [63]. For the third requirement, the STM must use the same SMR as STMCAS, and STM operations cannot require irrevocability or privatization.

To address the first requirement, we provide a hand-instrumented STM policy ("optSTM"), with which the programmer hand-instruments the data structure for STM. First, we modify xSTM by removing quiescence, replacing undo logging with

redo logging, and use the same SMR as STMCAS. This results in an implementation similar to SpecTM [80]. We then adopt the `Ownable` technique from Section 3.4.1, so that the programmer can control the mapping between locations and orecs. Finally, we employ a technique from previous work on library STM [113] and recent concurrent data structures [15], where the speculatively-accessed fields of each data structure node are expressed as a templated type. The template ensures that `txRead` and `txWrite` are called correctly, and via an `Ownable` argument, it lets the programmer ensure that the correct orec is associated with each field.

A thread can use optSTM to access a data structure while another thread accesses it with STMCAS [114]. To improve programmability and facilitate a more incremental transformation of optSTM code to STMCAS, we would like a thread to be able to execute a optSTM step after an STMCAS step (e.g., for a complex writing operation after a fast read-only traversal). To accomplish this, we introduce one new method, with a role similar to `checkVal`.

This new method, `inheritOrecs(`o, v`)`, must be called from within a optSTM transaction. Its arguments are the identity of an orec and a version number. If it observes that orec o is not acquired, and that $o.version = v$, it places o in the transaction's read set and returns `true`. Otherwise it returns `false`. The calling transaction can use the return value to decide whether to continue, or to exit the transaction and retry the preceding STMCAS step(s).

`inheritOrecs` relies on an important invariant: After a call to `inheritOrecs` by thread T with start time $T.start$ observes $o.version = v$, any change to data protected by o must also raise $o.version$ to a value $> T.start$. Otherwise, the update to o would have to be made by an operation that attained an end time before $T.start$. However, such an operation (whether optSTM or STMCAS) would

have acquired o before getting its release time, contradicting that $o.version = v$. Thus any subsequent validation by T, even using $T.start$ instead of v, re-asserts that locations protected by o have not changed since v was recorded by the prior STMCAS step. Given `inheritOrecs`, a programmer can transform *prefixes* of a optSTM transaction into STMCAS steps. In our evaluation, we apply this technique to red/black tree rebalancing and hash table resizing.

3.8 Safe Memory Reclamation

For completeness, Listing 7 presents the safe memory reclamation (SMR) algorithm used in our experiments. Note that exoTM itself does not require an SMR algorithm, and neither does the xSTM policy . However, both optSTM **??**nd STMCAS do.

optSTM and STMCAS are broadly compatible with most published SMR algorithms. Our decision to use this specific algorithm is guided by the following points. First, this SMR is compatible with all of the baseline algorithms in the evaluation section. Second, it does not require additional per-object metadata. Third, it does not have complex tuning knobs. Lastly, it employs the same hardware clock as exoTM. With regard to this last point, in our experiments we keep the SMR clock accesses separate from the exoTM clock accesses. In a production setting, we can save one clock access per exoTM operation, by using the SMR clock as the source for the start of the first optSTM or STMCAS step.

In the listing, a thread gets a start time, and publishes it, via the `enter` method. It clears its start time, both locally and in the global list, via `exit`. During an operation, threads keep a list of pending reclamations in the `_pending` set. When an operation completes, if it has pending reclamations, it gets a timestamp, and attaches that timestamp to its pending reclamations while transferring them to the

tail of the _frees queue. The only global communication comes when a thread calls _sweep. The thread must compute the minimum start time of all other threads. Since this overhead is linear in the number of threads, we amortize its overhead by only calling _sweep once per SmrThreshold times that _pending is not empty. In the experiments, we set this threshold to 1024. Finally, since elements in the queue are ordered by time, with the oldest first, the _sweep method can exit early as soon as it encounters an object whose associated deletion time exceeds the minimum of all thread start times.

Listing 6: Implementing xSTM from exoTM.

```
class StmThread // one per thread
1   singleton _orecTbl: Orec [NumOrecs ] // A table of orecs
2   singleton _threads: List ⟨StmThread⟩ // All threads

3   _checkpoint: RegisterCheckpoint // For restarting on abort
4   _exo: ExoTMThread // For exoTM functionality
5   _readset: Set⟨Orec*⟩ // used for validation
6   _undolog: Set⟨addr, val⟩ // used for rollback
7   _start: uint63 // Validity range for orecs; Initially 0
8   _alloc: Allocator // Buffers malloc and free until transaction end

9   constructor: _threads.insert(this)
    function txBegin(cp: RegisterCheckpoint)
10      _checkpoint ← cp ; _start ← _exo.assignStartTime()

    function _getOrec(addr: PointerType)
11      return &_orecTbl [rightShift(addr, 4) mod NumOrecs ]

    function txRead(addr: PointerType)
12      o ← _getOrec(addr)
13      while true do
14          pre ← _exo.checkStart(o) ; data ← atomicRead(addr)
15          if pre.locked ∧ pre.ver ≠ EndOfTime then return data
16          post ← _exo.checkStart(o)
17          if pre = post ∧ ¬pre.locked ∧ pre.ver ≠ EndOfTime then
18              _readset.insert(o) ; return data
19          if pre.locked then _abortAndRestart()
20          time ← _exo.assignStartTime()
21          _validate(_start) ; _start ← time

    function txWrite(addr: PointerType, data: value)
22      o ← _getOrec(addr)
23      while true do
24          ⟨ok, locked⟩ ← _exo.acquireStart(o)
25          if ok then
26              _undolog.insert(addr, *addr)
27              atomicWrite(addr, data) ; return
28          if locked then _abortAndRestart()
29          time ← _exo.assignStartTime()
30          _validate(_start) ; _start ← time

    function txCommit()
31      if _undolog.empty() then
32          _exo.clearStartTime()
33          _quiesce(_start) ; _alloc.reclaimFrees() ; _alloc.clear()
34          _readset ← {} ; return
35      _validate(_start)
36      time ← _exo.releaseOrecs(NewTime)
37      _quiesce(time) ; _alloc.reclaimFrees() ; _alloc.clear()
38      _undolog ← {} ; _readset ← {}

    function _validate(time: uint63)
39      for o in _readset do
40          if ¬_exo.checkVal(o, time).ok then _abortAndRestart()

    function _abortAndRestart ()
41      _undolog.undoAll() // atomic writes
42      _exo.releaseOrecs(IncTime)
43      _alloc.reclaimMallocs() ; _alloc.clear()
44      _readset ← {} ; _undolog ← {} ; _checkpoint.restore()

    function _quiesce(time: uint63)
45      for t in _threads do
46          while t._exo.readStartTime < time do wait()
```

Listing 7: Safe Memory Reclamation (SMR) API

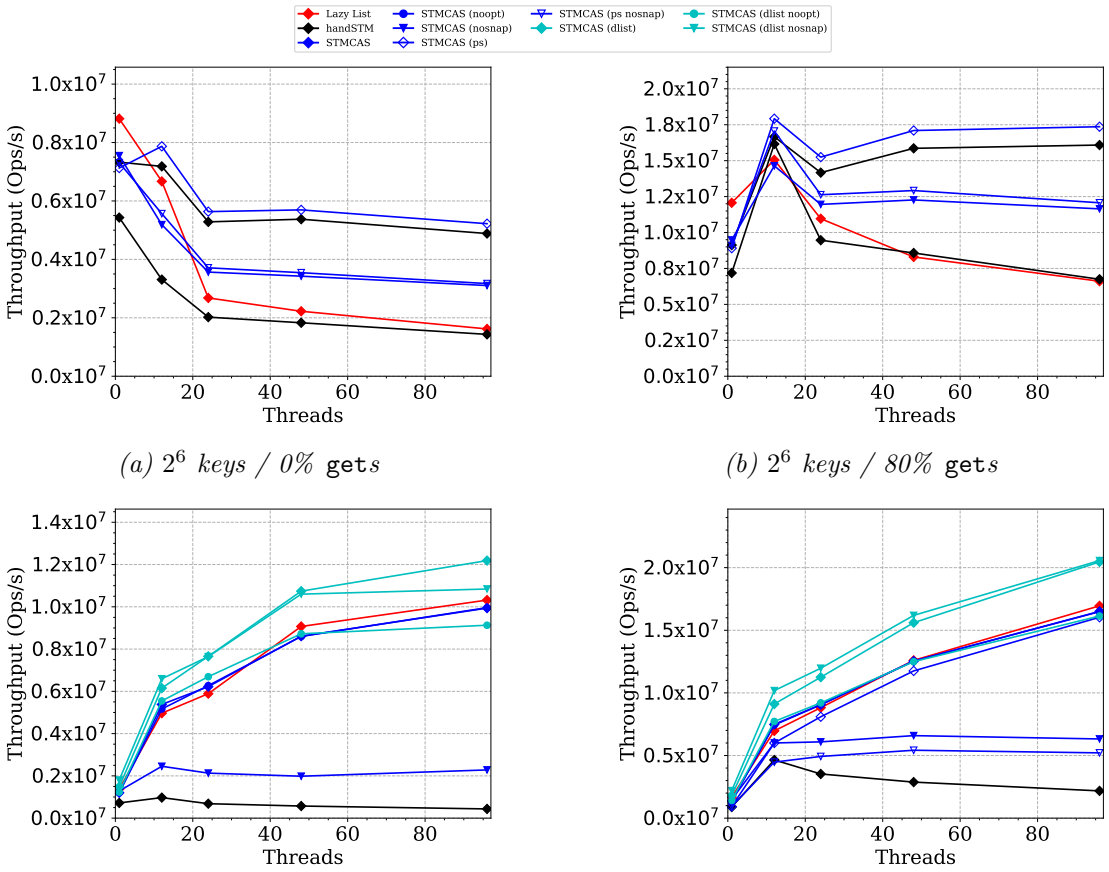
```
class SMRContext // one per thread
1     singleton _threads: List ⟨SMRContext⟩ // All threads
2     singleton _clock: Clock // Represents time as 63 bits
3     _ts: uint63
4     _pending: Set ⟨PointerType⟩
5     _frees: Queue ⟨PointerType, uint63⟩
6     _exits: uint63

7     function enter(): _ts.atomicExchange(_clock.inc())
      function exit()
8         _ts ← EndOfTime
9         if _pending = {} then return
10        time ← _clock.now()
11        for p in _pending do _frees.enqueue(⟨p, time⟩)
12        _pending ← {}
13        _exits ← 1 + _exits
14        if _exits < SmrThreshold then return
15        _exits ← 0
16        _sweep()
17    function reclaim(addr: PointerType): _pending.insert (addr)
      function _sweep()
18        minTime ← min(t._ts, t ∈ _threads)
19        while ¬_frees.empty() do
20            ⟨addr, _ts⟩ ← _frees.front()
21            if _ts ≥ minTime then return
22            free(addr) ; _frees.dequeue()
```

3.9 Performance Evaluation

To evaluate the impact of separating STM mechanisms from policies, we measure linked lists, binary search trees, closed addressing hash tables, and skip lists. Our baselines are the lazy list [108], a lazy list-based non-resizable hash table, Fraser's lock-free skip list [60, 115], David's external BST [109], and trees from PathCAS [15]. These generally represent the state of the art for each data structure [116, 115]. The only modifications we made were to integrate them into a common benchmark framework, to use the same SMR algorithm, and to reduce lazy list latency by replacing its pthread-based lock with a spinlock. We use the LLVM version [111] of TinySTM [61] as our STM baseline.

Our testbed has two Intel Xeon Platinum 8160 CPUs running at 2.10GHz (48 cores/96 threads) and 192GB of RAM. It runs Ubuntu 18.04.4 with Linux kernel 4.15.0-91. All code was compiled using llvm/clang 12 with –O3 optimizations. ExoTM uses `rdtscp` as its clock [106, 90]. We consider three policies based on exoTM. **xSTM** is the compiler STM algorithm from Section 3.6. We found that the LLVM TM plugin [111] had high overhead at transaction boundaries due to its use of lambdas. We developed a 300-line patch to eliminate this lambda-induced overhead. We then configured xSTM and TinySTM with 2^{20} orecs (each covering 32 bytes) and undo logging. **optSTM** is the hand-instrumented STM from Section 3.7. To match STMCAS, it embeds orecs in objects (Option 1 of Listing 2). Note that unlike xSTM, optSTM allows the programmer to skip instrumentation for `const` fields. **STMCAS** PO (per-object) embeds orecs in objects (Option 1); PS (per-stripe) uses a table of 2^{20} orecs (Option 2).

All data structures were pre-filled to 50% and all data points are the average of five 5-second runs. Keys are chosen with uniform probability. For lists and hash



(a) 2^6 *keys / 0%* `get`*s*

(b) 2^6 *keys / 80%* `get`*s*

(c) 2^{10} *keys / 0%* `get`*s*

(d) 2^{10} *keys / 80%* `get`*s*

Figure 3.1: Linked List performance

tables, the STMCAS snapshot threshold was 3. For the BST it was 1, and for the skip list we disabled snapshots to get a better performance. Note that our BST makes all modifications in-place. To handle reads concurrent with writes, it uses orec consistency instead of copying [117]. The STMCAS skip list, list, and hash table avoid consistency checks unless otherwise noted. Since its snapshot frequency is 1, the BST does not.

How Hard is STMCAS? As a coarse approximation of effort, we used `clang-format` on each data structure, then ran `cloc` [118] to measure lines of code

(our non-resizable hash tables use a common list adapter, which adds 30 lines). The results appear below. xSTM required the fewest lines of code. STMCAS generally had fewer lines of code than the baseline data structures, with many instances of code like line 37 of Listing 5. We believe that STMCAS is not harder than writing ad hoc CDSs (the baselines).

	Baseline	xSTM	optSTM	STMCAS
Linked List	113	67	71	164
Skip List	331	161	162	238
BST	282	98	113	208

Performance Under Low Contention: In our experiments, xSTM and TinySTM performed poorly. We only show their results for the BST (Figures 3.3a–3.3d). The difference between xSTM and TinySTM was negligible: quiescence dominated for both. Since our data structures do not require quiescence, optSTM scales better by avoiding it. The reasonable performance of optSTM holds for the BST, skip list, and hash table, but not for lists. This is a known problem with TM: low-level memory conflicts on list prefixes cause artificial validation failures.

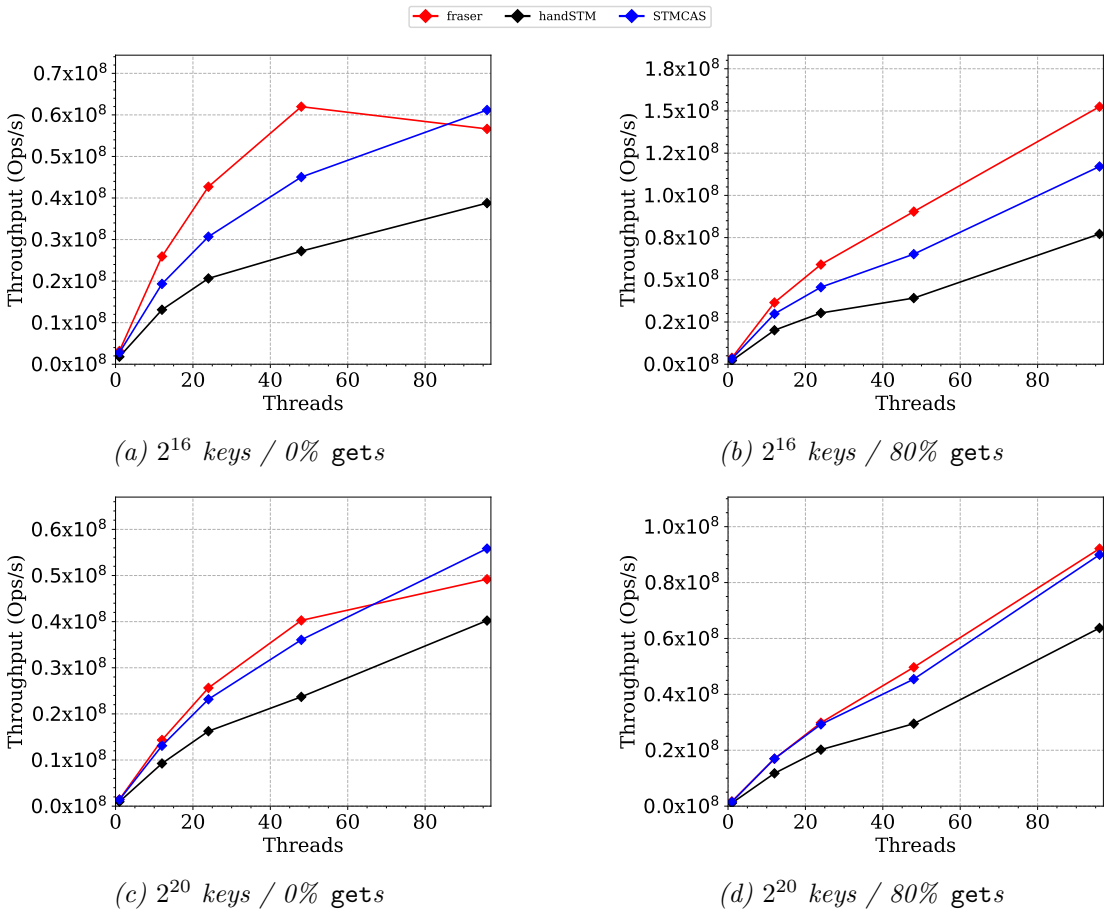
Comparing optSTM to STMCAS, we see that while both use orecs in similar ways, the avoidance of write logging, thread checkpointing, and read set validation give STMCAS a significant edge, except in the hash table. Our hash table is configured so that each bucket has 2 elements on average, and thus logging/validation overheads are negligible.

We now contrast STMCAS with the baseline data structures. For now, we focus on scalable workloads (i.e., not Figures 3.1a and 3.1b). First, we see that STMCAS outperforms the lazy list, and the doubly-linked STMCAS list performs even better. We find that STMCAS lists have lower latency: When consistency checks are elided, STMCAS does not need to mark and un-mark pointers, so there

is a small savings when visiting each node. The second difference relates to how modifications happen. Consider the DList: when `getLEQ` returns, `insert` does not check if a pair of nodes remain present and unlocked: instead `acquireVal` uses the expected orec value as a proof that the node is still present. This slightly reduces conflicts, and also reduces latency. A third, more subtle source of savings comes from the fact that synchronization occurs at the granularity of an entire node. Because DList nodes inherit from Ownable, each node is associated with a single orec protecting all of its structural links. This avoids per-field marking or validation and lets STMCAS acquire and validate ownership at the object level rather than the word level. While not the primary contributor to the latency differences above, this coarser (node-level) ownership reduces bookkeeping work and slightly shrinks the window in which conflicts may arise, providing an additional practical benefit.

The hash table shows one weakness of exoTM policies: the `rdtscp` instruction introduces latency at the beginning of every step, and again when completing a write.

The skip list results are more nuanced than the list. First, the latency of an individual `insert` or `remove` is lower in the lock-free baseline, which uses `CAS` to modify values directly, instead of acquiring orecs and then modifying values. Second, since conflicts are less common, snapshotting (which adds instructions to check orec values) adds latency but does little to increase scalability. Third, the baseline does not incur `rdtscp` overheads on every operation. Together, these properties place STMCAS at a disadvantage for `get`. However, the STMCAS `insert` and `remove` steps can exploit blocking to be more efficient: They can acquire a node and all of its predecessors before making any updates, which sacrifices lock freedom but avoids re-computing predecessors when a linearized modification conflicts in



(a) 2^{16} *keys / 0%* `gets`

(b) 2^{16} *keys / 80%* `gets`

(c) 2^{20} *keys / 0%* `gets`

(d) 2^{20} *keys / 80%* `gets`

Figure 3.2: Skip List performance

the index. Thus at high thread counts and high write ratios, STMCAS can match the lock-free skip list's performance.

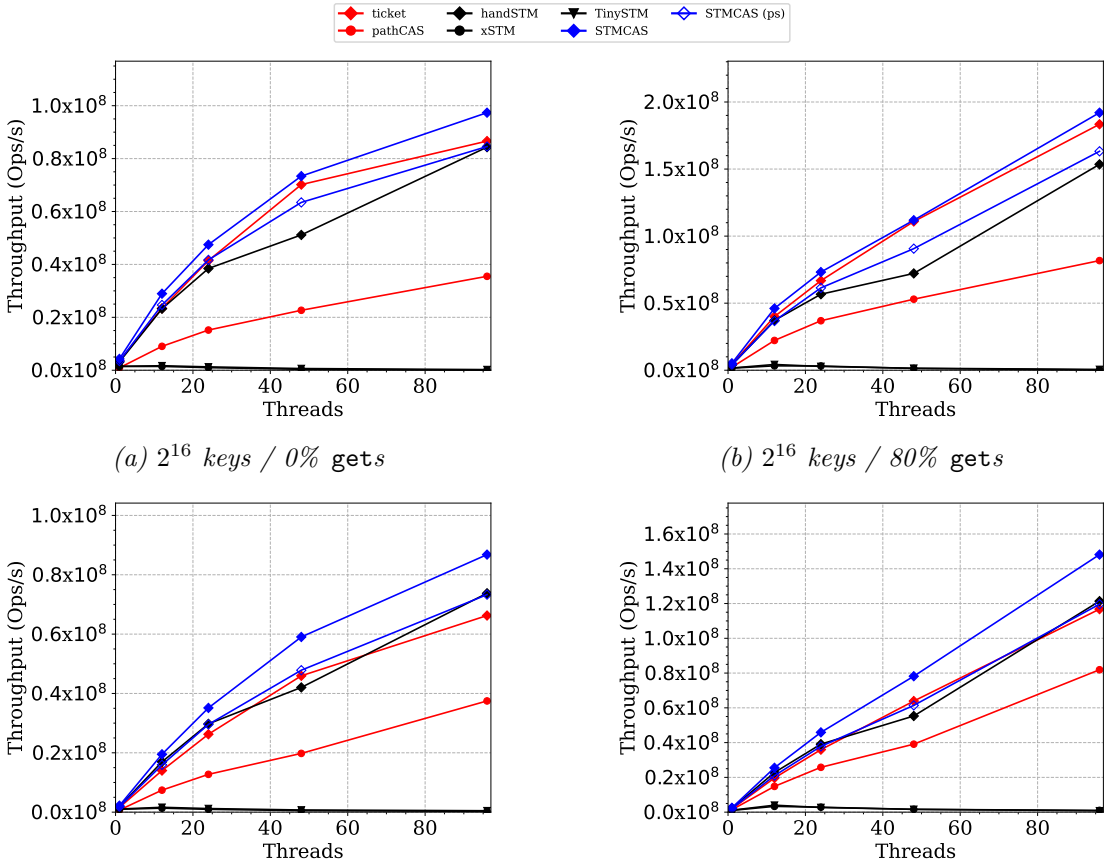
Our STMCAS BST moves keys and values, not nodes, when swapping a node with its successor during `remove`. Thus every visited node must be included in the snapshot. From the preceding discussion, we know that each orec check introduces latency, and thus the BST traversal is less efficient than the skip list. However, there are analogous costs for the baseline BST, which uses Ticket locks, and PathCAS. In the end, the biggest gains come from snapshotting, and from the ability to linearize operations without validation in cases analogous to lines 3 and 16 of Listing 5, and

line 3 of Listing 4.

Performance Under High Contention: Lists with a 2^6 key range exhibit different characteristics than the other workloads. In low-contention experiments, PO versions outperformed their PS counterparts, because of the locality benefits of co-locating orecs with data. However, in high-contention settings, there is a synergy between skipping orec checks and placing orecs in a separate table: it reduces cache misses. We also see that snapshotting did less to help under high contention. This is a consequence of contention *invalidating* snapshots: In Listing 5, taking a snapshot causes an orec check, which can cause the step to restart. Alternatively, the step can skip that snapshot rather than restart. However, doing so may lead to there being too few valid snapshots from which a failed step can resume.

Other Considerations: On one hand, we find that STMCAS simplifies implementing complex CDSs. We already saw that the DList was easy to implement and performed well. Similarly, we include a closed addressing, resizable hash table in our experiments. Our design resizes the table lazily, like the Liu hash [119]. In the common case, operations read one more location than their non-resizable counterpart. The performance overhead of one additional read and orec check is negligible, but the code increases from 194 to 321 lines.

On the other hand, splitting operations into too many steps was detrimental. For example, we created a doubly-linked skip list (not shown) whose `insert` and `remove` used a logarithmic number of small writing steps, instead of one big step. E.g., when inserting, each step traverses backward and upward to connect to the predecessor at the next index layer). The technique added latency (e.g., `rdtscp` and releasing and re-acquiring orecs) without increasing scaling. As future work, we plan to study if skew alters this tradeoff.



(a) 2^{16} *keys / 0%* `gets`

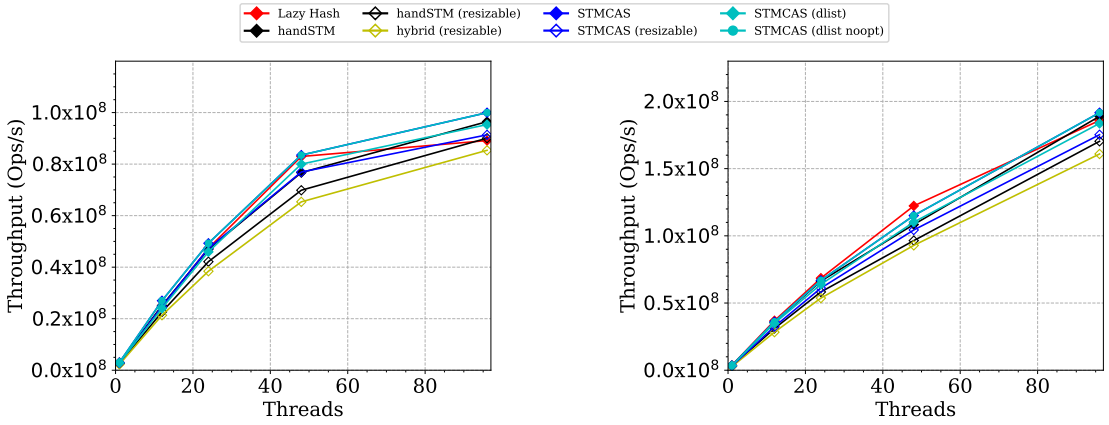
(b) 2^{16} *keys / 80%* `gets`

(c) 2^{20} *keys / 0%* `gets`

(d) 2^{20} *keys / 80%* `gets`

Figure 3.3: Binary Search Tree performance

Using STM with STMCAS: Figures 3.4 and 3.5 provide an opportunity to assess how STMCAS and STM can interoperate, on a resizable hashtable and balanced tree, respectively. In the hash table, each operation computes an array index, and then checks the list at that index in the "active" table. If the list is nonexistent, the operation performs a resize from the "old" table, and then restarts. Resizing entails atomically extracting a list from "old" and migrating the entries into two lists in "active". This is accomplished via reads of (immutable) keys and writes of predecessor and successor pointers. The code is simpler in optSTM than in STMCAS, since the programmer does not have to explicitly interact with the



(a) 2^{20} *keys / 0%* `gets` *(b)* 2^{20} *keys / 80%* `gets`

Figure 3.4: Hash Table performance

orecs. But since the lists are typically small, and conflicts rare, the behavior of optSTM and STMCAS is indistinguishable. In this setting there is no benefit to combining STMCAS and optSTM: extracting a prefix of the optSTM transaction and turning it into an STMCAS step does not save enough to overcome the second call to `rdtscp`.

In the balanced tree, we find that combining optSTM and STMCAS is more profitable. The pathcas_avl tree is the balanced tree implementation from [15] (roughly 700 lines of code). The optSTM version is derived from a sequential red/black tree, in 240 lines of code. The STMCAS version uses the same `get` as the unbalanced tree. For `insert` and `remove`, after a read-only step identifies a node to remove and its successor, or the parent of the node to insert, we employ an extra read-only step to compute the full set of color changes and re-balances. We then use a (potentially large) writing step to make all changes as a single MCAS-like step. This complexity adds 156 lines of code. Lastly, our hybrid approach uses STMCAS for the initial read-only search, and then uses an STM transaction to do the remaining work. This only adds 35 lines of code.

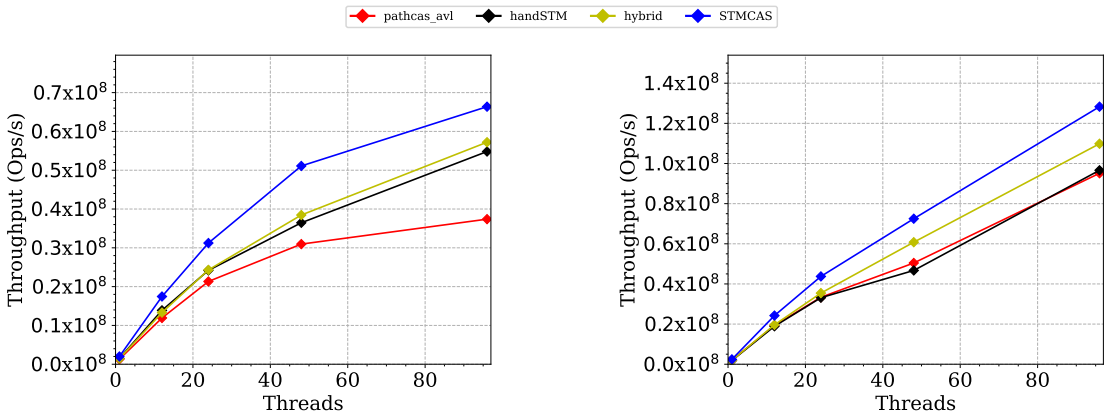
The hybrid implementation closes almost half the gap between STM and STMCAS. While redo logging is still present, most operations do only a small amount of rebalancing, so the log remains small. Similarly, validation in the final STM operation is usually small, since it does not involve nodes from the read-only traversal. Instead, overheads like thread checkpointing (`setjmp`) become more pronounced. This suggests that with further optimization, the hybrid implementation could close more of the gap, without the same increase in complexity that STMCAS suffers.

We caution against broad conclusions about the performance of PathCAS/Optik versus STMCAS/optSTM. We believe that some of our techniques, like snapshotting, could accelerate prior work. Furthermore, prior work did not have as much room for hand optimization.

3.10 Related Work

Elastic transactions [120] and early release [59] were early attempts to let programmers avoid validation for locations that become irrelevant to an operation's consistency. Their strategy is the reverse of ours: rather than asking the programmer to "hold on" to locations that need to be kept consistent, they encourage the programmer to "let go" of locations that do not need to be kept consistent. Such an approach is more appropriate for a general-purpose STM, but the lack of awareness of the mapping from locations to orecs makes it difficult to use these techniques well in general-purpose code.

The idea of splitting transactions into a read-only (traversal) phase followed by a write-only (commit) phase dates back at least to Consistency Oblivious Programming [121] and Optimistic Transactional Boosting [122]. Those approaches delegate most of the work to programmer-defined mechanisms (like a programmer-



(a) 2^{20} *keys / 0%* `gets` *(b)* 2^{20} *keys / 80%* `gets`

Figure 3.5: Balanced Tree performance

defined validation procedure to run before the commit phase). We believe that explicit reasoning about orecs in exoTM is less error-prone.

Another related area focuses on splitting hardware transactions to avoid hardware capacity limitations. While some of the mechanisms are similar to our work, these ideas [123, 124, 125, 126] are not applicable to systems without HTM. It would be interesting to explore whether it is possible to build a fast STMCAS-like policy on top of a hardware TM mechanism.

MCMS [39] and PathCAS [15] blur the line between MCAS and STM, while lock-free locks blurs the line between locking and STM [127]. However, all three emphasize non-blocking progress, which causes higher latency when releasing ownership, and requires a helping mechanism. In addition, none of these use a clock to simplify versioning. We show that the clock is essential to enabling snapshots and splitting an operation into steps. We look forward to exploring if ideas from these works can be integrated with STMCAS, especially for non-blocking progress. We also expect that ideas from STMCAS, like snapshotting, could improve these

approaches.

Optik [128] takes an approach to CDS design that is similar to ours, approximating version information via ticket locks. While Optik did not provide snapshots, we see no reason why our approach would not be applicable to it. Likewise, our multi-step approach seems possible in Optik. As we saw in the evaluation, the use of a clock gives exoTM an edge versus the ticket BST, because we can ensure consistency in a read-only traversal without double-checking versions. In addition, exoTM/STMCAS makes it easier to move from single locks to MCAS-like operations, and to integrate STM for uncommon code paths. However, OPTIK does not require all locks to be released at the end of an operation. It would be worthwhile to study if this difference affords it performance opportunities that exoTM lacks.

3.11 Conclusions and Future Work

This chapter showed that the core synchronization mechanism of STM can be extracted from the rest of a full-fledged STM system. The chapter then showed that the resulting "exoTM" could be used with other synchronization interfaces ("policies"). This culminated in the STMCAS policy, which provides compelling performance. An added benefit is that STMCAS is compatible with STM, so that a programmer can start with an easy but correct STM data structure, and then optimize incrementally.

One of the key features of STMCAS is that orec values are visible to programmers, and can be used to construct proofs that invariants of a failed data structure traversal still hold. As future work, we plan to study if this can be applied to prior work. There is also much potential in leveraging exoTM's programmer-specified mapping from locations to orecs to increase the number of values stored in each

data structure node, without false conflicts. Lastly, as byte-addressable non-volatile memory favors transactional programming styles, there is an opportunity to extend exoTM to support persistent data structures.

Chapter 4

Harmony: Composable Transactional Data Structures

This chapter extends the architectural decoupling principles from Chapter 3 to address composition across multiple data structures. Transactional data structures are a subclass of concurrent data structures that support composition. That is, they are augmented so that programmers can combine a sequence of method invocations by a single thread into a coarse-grained "transaction". Each transaction appears to run in isolation, without interference from concurrent transactions. To achieve this, transactions must be able to detect and recover from situations where method invocations from concurrent transactions have semantic conflicts. Conflict detection is achieved by way of rules about how data structure methods interact with synchronization metadata, transaction metadata, and program state. These rules are encapsulated in a transactional data structure run-time management system. A system's rules can impact latency, throughput, and generality.

This chapter introduces Harmony, a transactional data structure system that

explores a novel design point. Harmony carefully orchestrates two categories of metadata: one for detecting low-level memory conflicts during individual data structure operations, the other for detecting conflicts among transactions. Importantly, both types of metadata are embedded in the individual nodes of data structures, and are protected by the same synchronization technique. This approach avoids false conflicts among transactions, avoids unnecessary coupling within the implementation, and delivers a high degree of control to the data structure designer.

On account of this design, Harmony enables programmers to implement data structures of arbitrary complexity, including skip lists, resizable arrays, deques, and resizable hash tables. These data structures easily support complex features, such as read-only and mutating range queries. Furthermore, Harmony keeps overhead low and avoids scalability bottlenecks. In evaluation on microbenchmarks and the TPC-C benchmark suite, Harmony outperforms the current state of the art, often by a significant margin.

4.1 Background and Motivation

The preceding discussion established the need for composable transactional operations across data structures while avoiding the tight coupling of prior approaches. Before diving into Harmony's design, we provide additional background on the challenges of composition and the landscape of prior solutions.

Concurrent data structures are specially designed so that they behave correctly even when many threads invoke their methods simultaneously. The most common correctness criteria for concurrent data structures is linearizability [26], which requires each method to appear to happen at a single point in time between its invocation and response (its "linearization point"), and for the result of a history

of concurrent method executions to be equivalent to the result of some sequential history in which there is a total order on all linearization points. Linearizability has proven to be a powerful correctness criteria: There are many blocking and non-blocking data structures that scale well and are linearizable [8].

Linearizability is defined in terms of the invocation and response of a *single* method. Thus it is not straightforward to compose two or more operations on concurrent data structures in a manner that is linearizable. Suppose thread T performs method M_1 on linearizable data structure D_1 and then passes the result to method M_2 on linearizable data structure D_2. (As a simple example, one may suppose that M_1 is a withdrawal from one bank account, and M_2 is a deposit into another account). Since D_1 is linearizable, M_1 must appear to happen between the moments of its invocation and its response by T. Necessarily, the response from M_1 must precede the moment when M_2 is invoked. Herein lies the problem: The definition of linearizability does not forbid other threads from invoking an arbitrary number of operations (such as interest calculations) on D_1 and D_2 between the moment when T receives M_1's response and when it invokes M_2. The fact that every method of D_1 and D_2 linearizes is insufficient to ensure that the sequence of method invocations and responses by T runs without interference by other threads.

Drawing inspiration from both traditional database systems [129, 130] and transactional memory (TM) [9, 10], *transactional data structures* (TDSs) address this problem. In a TDS-based system, programmers *begin* a transaction, *invoke* data structure methods, and then try to *commit* the transaction. The methods of TDSs are specially designed to be compatible with transactions. In particular, while it is reasonable to think of each data structure operation as linearizing, a transaction's effects on a data structure are not visible to other threads until the

enclosing transaction commits. When a TDS system allows dynamic control and data flows within a transaction, it is possible for concurrent transactions to conflict (i.e., the methods executed by one transaction cannot commute with the methods executed by another), in which case one or more may need to abort and retry.

Part of the appeal of TDSs is that they provide developers with an intermediate point between the rigid structure of databases and the ad hoc nature of TM. Like TM, these systems allow programmers to create arbitrary transactions, with dynamic control flows, that operate over a variety of data structures. Like databases, they restrict programmers to a set of highly-optimized data structures, which make it easier to keep transaction management costs low. However, TDSs introduce a host of design challenges, including:

1. How to synchronize individual method invocations, so that each method of a TDS is as scalable as an equivalent concurrent data structure method?
2. How to detect transaction conflicts efficiently?
3. How to support complex data structure operations?
4. How to manage program data upon commit and abort, considering that commits and aborts may happen long after individual methods linearize?

The primary focus of most prior work has been on addressing the first two challenges. To address these challenges, the designers of TDS systems provide a collection of types that are understood by the TDS system, but manipulated by the data structure designer. These types are often referred to as metadata, since they are largely orthogonal to the information used by the program to compute its results. The methodology that a TDS system introduces for using this metadata reifies its solution to the above challenges. Below, we briefly summarize the two main approaches of past work.

Coupling Synchronization and Transaction Conflicts: This approach provides one kind of metadata that can be used for low-level synchronization of individual data structure operations, as well as high-level expression of transaction-level conflicts. Example systems include TDSL [13], STO [131], LFTT [132], and Medley [133]. The primary purpose of the metadata is to coordinate data structure operations to avoid races. While the details differ, these systems all focus on data structure synchronization techniques that are fundamentally optimistic (e.g., by way of versioned locks). Since *some* version is already associated with each logical entry in each data structure, these systems then use that same versioning for detecting transaction-level conflicts. The main benefits of this approach are efficiency and generality: Transaction overheads are low, since only one set of metadata needs to be managed; and programmers can use the metadata at a fine level of granularity (e.g., attaching instances to individual data structure nodes), so that high-level transaction conflicts appear as low-level memory conflicts. However, non-conflicting operations issued from concurrent transactions can misinterpret data structure synchronization-related conflicts as transaction-level conflicts, resulting in some transactions aborting and retrying unnecessarily.

Out-of-Band Transaction Conflict Metadata: In systems like Boosting [134], Proust [135], and Conflict Abstractions [136], high-level "abstract" metadata coordinates transactions at the granularity of the semantics of operations, while data structures are synchronized via separate metadata. These systems can use any off-the-shelf concurrent data structure by wrapping its methods with code that interacts with the metadata. Necessarily, the transaction metadata is stored separately from the concurrent data structure. An appealing characteristic of this approach is that individual data structures can use different techniques for

achieving concurrency (such as fine-grained locks or lock-free techniques). However, in this approach it can be difficult to efficiently express the conflict conditions for complex operations in terms of the metadata. For example, we are not aware of a disjoint-access parallel [137] metadata design when range queries over sets of floating-point numbers can run concurrently with insertions and removals.

The approach to the first two questions influences how a TDS system addresses the remaining two questions. In the *coupled* approach: complex data structure operations can be supported by managing ownership and versioning of individual values or pointers in the data structure; the TDS system is likely to require the programmer to wrap program data in special types that let the system perform undo/redo when a transaction commits or aborts. In the *out-of-band* approach: it can be challenging to support complex data structure operations (see above); while the data structure must provide semantic undo functions (e.g., `insert(x)` pairs with `remove(x)`) to handle aborts, the system does not place requirements on the management of program data.

This chapter introduces Harmony, which seeks to strike a balance between the two approaches. Harmony uses separate metadata fields for data structure synchronization and for transaction management, but these two kinds of metadata work together, and are co-located within the data structure. In Harmony, transaction management metadata is not abstract or semantic. Its purpose is to allow the data structure designer to separate the temporal ownership and versioning of a datum, needed for preventing races, from the ownership and versioning needed for detecting conflicts among transactions. Since the metadata is placed within the data structure, Harmony is able to express complex operations' conflict conditions efficiently. With regard to managing program data, Harmony more closely resembles

the out-of-band approach: It does not place any rules on how program data is managed, instead letting each data structure define how operations should clean-up upon abort or commit.

During the design and implementation of Harmony, two issues that have received insufficient attention in prior work were identified. The first is that capturing non-existence of an element in a set or map is subtle: Prior techniques can result in false conflicts, starvation, or an inability to generalize to complex data types. The second is that the manner in which the transaction system cleans up speculative writes to a data structure upon commit/abort can lead to transient inconsistencies. Harmony addresses the former through three compatible techniques for expressing nonexistence, and the latter through an innovative use of coroutines.

The remainder of this chapter is organized as follows. Section 5.6 discusses prior transactional data structure research. Then, Section 4.3 presents an overview of Harmony, including its transaction management metadata and run-time operations. Sections 4.4 and 4.5 introduce solutions for non-existence and transient inconsistency during cleanup. Section 4.6 discusses correctness, and then Section 4.7 briefly sketches the implementation of several data structures to show that Harmony can support complex use cases (such as range queries and resizable hash tables). Section 4.8 shows that Harmony provides best-in-class performance on microbenchmarks and the TPC-C benchmark suite. Finally, we draw conclusions and discuss future work in Section 4.9.

4.2 Related Work

TM was originally proposed as a hardware (HTM) [9] and then software (STM) [10] technique for implementing concurrent data structures. As STM became increas-

ingly practical [59, 138, 60], attention turned to the role it could play in composing multiple operations into a single coarse-grained transaction [11]. However, experiences using both HTM and STM in benchmarks and real-world code indicated that TM, in general, was too conservative, treating every memory conflict as a true conflict [139, 140, 141, 142, 79].

Transactional Boosting [134] was the first attempt to emphasize transactions for composition of data structure operations. It manages transactional metadata (abstract locks) that are separate from the underlying "black box" data structures. Boosting requires a "semantic" undo mechanism in case of abort (e.g., `remove`↔`insert`). Proust [135] extends boosting with a lazy mechanism where operations are executed on shadow copies of an object, and replayed on the shared copy only when the transaction is guaranteed to successfully commit. This helps to support operations with tricky semantics, e.g., priority queue insertion and extract-min.

TDS systems that use a single type of metadata build upon attempts to inject the notion of semantic conflicts into the TM frameworks themselves [143, 59, 90]. For example, early release [59] allows a transaction to remove memory locations from its read-set if it knows that those locations will no longer raise semantic conflicts. The Transactional Data Structure Library (TDSL) [13] greatly expands upon this idea. For example, its ordered map superimposes Fraser's skip list [60] with a TL2 [42] STM-based, early release-optimized singly linked list. Optimistic Transactional Boosting [122] is a similar technique, which integrates transactional metadata into "white box" data structures. Elastic transactions [120] took a different approach to normalizing and regularizing when entries enter a transaction's read set, so that an STM could avoid conflicts and support greater composability. Software

Transactional Objects (STO) [131] is another work in this vein. STO defines the data types that a low-level transaction can use (e.g., `TBox<int>`), and then introduces predicates and other means for expressing semantic conflicts (or the lack thereof). For example, one STO type enhances an earlier idea from Ruan et al. [90] to allow increments and decrements of a shared counter to commute with each other, but not with reads of the counter value.

The most recent TDS systems have focused on progress and persistence. The Lock-Free Transactional Transform (LFTT) [132] carefully defines the semantics of operations, so that a transaction can *publish* its in-progress actions into data structure nodes, thereby facilitating helping. OneFile [144] translates transactional data structures to non-volatile byte-addressable memory. Most recently, Medley [133] achieved nonblocking progress and persistence. The key innovation in that work was to leverage existing linearization points of nonblocking data structures as the hook for expressing pending/transactional operations, and superimposing multi-word compare-and-swap [138] atop these linearization points.

An orthogonal line of research aims at enabling range queries in highly optimized concurrent data structures [145, 146, 147, 148]. However, none of these approaches allows range queries to be executed within a transactional context, and they also do not support bulk updates.

4.3 The Harmony TDS System

In this section, we discuss the design and implementation of Harmony. There are three key variable types. `HMeta` is the type for metadata that Harmony uses to detect when transactions conflict over a specific node in a data structure. `HTx` provides per-thread storage for the state of an in-progress transaction. Finally,

`notexist` is a global table of version numbers, whose use is discussed in Section 4.4.

4.3.1 Data Structure Synchronization

Since every TDS is also a concurrent data structure, the first obligation of a TDS system is to specify how data structure synchronization is achieved. Harmony does not mandate that any specific synchronization technique be used. Instead, it lets each data structure take responsibility for its own synchronization. For a given data structure, we use `S` to indicate the synchronization mechanism it uses.

A key idea in Harmony is that `HMeta` objects must be accessed (and in some cases modified) *atomically with* operations on the data structure nodes with which they are associated. Thus for a synchronization mechanism `S` to be compatible with Harmony, it must support in-place atomic multi-word updates. Several modern synchronization frameworks provide this support [39, 19, 128], as do many transactional memory implementations.[1] To capture the intuition that the `HMeta` instances for a data structure are synchronized via that data structure's choice of `S`, we represents the types of fields of `HMeta` as instantiating a template `field` with `S` (Listing 8).

To give an example, assume transaction T is running a TDS operation O that updates node n of data structure D, and D uses synchronization mechanism `S'`. There will be a `HMeta` object h associated with n, so the `owner` field of h will be of type `field<S', &HTx>`. Therefore, operation O can be synchronized according to S' for both its accesses to data structure fields of n and also fields of h. This means, for example, that as part of O, using only S', the data structure designer can atomically update `n` and set h's `owner` to `T`. This way, Harmony achieves

[1]Note that if a data structure were protected by a coarse grained lock, it would also meet this requirement, though it would be unlikely to scale.

the intended balance between the two aforementioned metadata management approaches. Similar to the *out-of-bound* approach, Harmony separates the low-level data structure synchronization metadata (defined by `S`) from the high-level transactional management metadata (i.e., fields of h). Similar to the *coupled* approach, the fields of h are embedded in the data structure and express conflicts at the granularity of memory, not abstract operations.

It is worth noting that Harmony may need to access fields of `HMeta` instances from *outside* of data structure code. For example, in Section 4.3.5 we discuss transaction-level events like validation. This validation may need to check the fields of the `HMeta` instances accessed while the transaction executes. We do not want these accesses to require Harmony to know details of each choice of `S`. Instead, we place one additional requirement on `S`: It must allow for the variables it protects to be read from unsynchronized code, without causing a race. In C++, this simply means that each field must be accessible via an `atomic_ref` [107], and is represented by the use of `field_base`, which is *not* templated on `S`.[2]

4.3.2 Memory Safety

In order to scale, most modern concurrent data structures rely on optimistic reads. In languages like C++, these can lead to errors unless in-flight operations can prevent the reclamation of locations that might still be accessed by (doomed-to-abort) speculative operation attempts. Safe memory reclamation (SMR) is a deferral technique for achieving this goal, and is used by almost every synchronization framework designed over the last decade [127, 39, 128, 15, 40, 19]. SMR is also

[2]Genericity over `S` does not imply that data structures require direct and unchecked access to fields of `HMeta` instances. Proper encapsulation and good software engineering practices are still easy to support.

Listing 8: Per-node metadata and global variables.

```
   class HMeta
1      owner : field⟨S, &HTx⟩
2      semantic : field⟨S, u64⟩
3      structural : field⟨S, u64⟩
4      num_coros : field⟨S, u32⟩
5      removing : field⟨S, bool⟩
6      undo_logged : field⟨S, bool⟩

   // notexist is an out-of-band table for detecting when a
   // node that was not present becomes present. See Section 4.4
7  const notexist_size : 2^24
8  global notexist : field_base⟨u64⟩[notexist_size]
```

used by all of the works discussed in Section 4.8.

While it is safe to assume that each `S` will provide its own SMR, doing so is not safe. The reason is that a doomed-to-abort transaction must be able to read `HMeta` fields from deleted nodes, in order to detect the need to abort. This means that the dynamic scope of a "safe" SMR region must align with the boundary of a transaction, not the boundary of a single data structure operation within the transaction. Rather than have Harmony manage a plurality of SMR regions, one per `S`, we require all instances of `S` to agree upon a single SMR implementation. Harmony itself manages the SMR, which we represent by keeping per-thread SMR state in the `HTx.smr` field (Listing 9). We assume, without loss of generality, that `HTx.smr` ensures memory is not recycled or returned to the operating system while a doomed transaction still might hold a reference to it. When we discuss transaction-level events (Section 4.3.5), SMR protection will be achieved through a dynamic scope bounded by `smr.enter()` and `smr.exit()`.

4.3.3 Transactional Versioning and Ownership

Given a synchronization mechanism `S` and a guarantee, through SMR, that `HMeta` instances are not reclaimed during a transaction's execution, we can now discuss

`HMeta` (Listing 8) in proper detail. For now, we are only concerned with the `owner` and `semantic` fields, which provide basic ownership and versioning. In Section 4.4, we will discuss a more refined versioning strategy, which makes use of `structural`, as well as the global `notexist` table. Discussion of the remaining fields on lines 4– 6 is deferred until Section 4.5.

The role of `HMeta` is to provide a fine-grained way of letting transactions detect conflicts. In many regards, this role is similar to that of ownership records (orecs) in software TM [42]. Prior STM research has considered a variety of approaches for mapping program data to orecs [61]. While similar flexibility is possible in Harmony, we focus on a simple design: In linked data structures, we place one `HMeta` object in each node. In arrays, we associate one `HMeta` with each entry (Section 4.7.3 discusses an alternative).

The `HMeta` type includes an `owner` field, as well as a `semantic` version. At a high level, we say that any transaction wishing to modify the state of a data structure node or array entry must be the exclusive owner of the associated `HMeta`. This is achieved by using `S` to atomically transition the value of `HMeta.owner` from empty to a reference to the transaction's `HTx` instance. We further require that before a transaction's first change to program data protected by an `HMeta` instance, it must increment `HMeta.semantic`. Incrementing the `semantic` version indicates that the logical state of the node has changed. This can include modifying values (e.g., on `update`) or marking `HMeta.removing` to indicate that the node is no longer logically present. Finally, *recording* the `semantic` version allows a transaction to detect, at any time, when the logical state of a node has changed. Note that the `semantic` version need *not* be updated when a node's pointers are changed, or when an array is resized.

In summary, during a data structure operation, before interacting with program data, the programmer interacts with the corresponding `HMeta` object as follows:

1. If the correctness of the operation depends on being able to *write* to program data, then the operation must take ownership of the `HMeta` object by changing the `owner` field from empty to a reference to the thread's `HTx`, and incrementing `semantic`.
2. If the correctness of the operation depends on the value read from program data, then the operation must ensure the `HMeta` object's `owner` is not some other `HTx`, and must log the version (`semantic`).
3. Otherwise, the `HMeta` object can be ignored.

We argue that these rules suffice to ensure that any two operations that do not commute will conflict. Observe that in any linearizable data structure, immediately before some operation O, all threads agree upon the set of locations that O would need to access in order to linearize. Any non-commuting operation must be guaranteed to manifest a conflict on at least one of those locations. Thus as long as the programmer has associated some `HMeta` object with each location, there will be sufficient ownership and versioning to detect conflicts with high precision.

It remains to be shown that the conflict will cause at least one transaction to abort. As in prior work, there are three types of conflicts: read-after-write, write-after-write, and write-after-read. The first two will be detected when the second operation discovers that `owner` is set. The third condition is detected by the `validate` function, which performs version-based validation (Cf. optimistic concurrency control [149]).

Listing 9: Per-thread metadata and supporting methods.

```
   class HTx
1  |  status : Inactive | Active | Aborted
2  |  reads : map⟨field_base⟨u64⟩*, u64⟩
3  |  coros : vector⟨handle⟩
4  |  smr : safe_memory_reclaimer

   function log_semantic(self: &HTx, m: &HMeta) → bool
5  |  let new_v = m.semantic
6  |  let old_v = tx.reads.get(&m.semantic); if old_v: return old_v == new_v
7  |  tx.reads.set(&m.semantic, new_v); return true
   function inc_semantic(self: &HTx, m: &HMeta) → void
8  |  ++m.semantic
9  |  let &old = self.reads.find(&m.semantic); if old: ++old.val
   function log_struct(self: &HTx, m: &HMeta) → bool
10 |  // same as log_semantic, but using m.structural instead of m.semantic
   function inc_struct(self: &HTx, m: &HMeta) → void
11 |  // same as inc_semantic, but using m.structural instead of m.semantic
   function log_notexist(self: &HTx, ki: u64) → bool
12 |  let idx = hash(ki) mod notexist_size
13 |  // same as log_semantic, but using notexist[idx] instead of m.semantic
   function inc_notexist(self: &HTx, ki: u64) → void
14 |  let idx = hash(ki) mod notexist_size
15 |  // same as inc_semantic, but using notexist[idx] instead of m.semantic
```

4.3.4 Per-Thread Metadata

Listing 9 introduces the per-thread `HTx` type, which is used to track the progress of transactions. Each thread re-uses its `HTx` for each transaction it executes, tracking its current state in the `status` field. The `Inactive` and `Active` states aid in debugging (i.e., when a transaction is started while another is still active). `Aborted` is used when a transaction discovers that it cannot succeed, but has not yet rolled back and retried. This is necessary because Harmony does not explicitly checkpoint and restore threads, instead leaving that up to the user of Harmony TDSs.

For the time being, it suffices to understand `HTx.coros` as containing sufficient information for releasing ownership of `HMeta` objects when a transaction commits or aborts. (More detail is provided in Section 4.5). Also, we already discussed how the `smr` field is used for safe memory reclamation. The remaining field in `HTx` is the `reads` field, which will be our focus in the remainder of this subsection.

Recall from Section 4.3.3 that a transaction records `semantic` (and eventually `structural` and `notexist`) versions, and later checks those versions to detect write-after-read conflicts. Designers of TDS operations should strive to minimize the number of versions that must be recorded to detect conflicts, since all recorded versions must be checked every time a transaction validates. There are two additional concerns, which lead to our presentation of six methods for managing how versions are tracked. We will focus on `log_semantic` and `inc_semantic`; the other four functions are closely related.

To reduce the cost of validation, `log_semantic` first checks if this specific `HMeta.semantic` was already recorded. If so, it ensures that the observed version and the current version match, and returns false otherwise. A false return notifies the calling TDS operation that the calling transaction must abort, because a concurrent transaction changed a `semantic` version upon which this transaction depends. A return of true indicates that the version has not changed, or has not been observed before. It also indicates that the location will be checked on all future validations.

Of course, a transaction also must not conflict with itself. If a transaction owns an `HMeta`, attempts to re-take ownership should succeed. In order for a TDS operation to change a `semantic` version, it must use `inc_semantic`. `inc_semantic` increments the version and then checks `Tx.reads` to see if an earlier version was read by the transaction. If so, the version number used for validation will also be incremented. This prevents a transaction from recording a version, incrementing it, and then mistakenly self-aborting on its next call to `validate`. Note that if an intervening transaction also incremented the version, the our choice of incrementing the log entry, instead of writing the new `semantic` version, ensures that a future

Listing 10: Transaction-level management routines.

```
    function begin(self: &HTx) → void
1       self.status = Active; self.smr.enter()
    function validate(self: &HTx) → bool
2       if self.status == Aborted then
            return false
3       for [addr, val] in self.reads do
4           if *addr ≠ val then self.status = Aborted; return false
5       return true
    function rollback(self: &HTx) → void
6       self.coros.reverse().each(|c| c.resume())
7       self.status = Inactive; self.smr.exit(); self.coros.clear(); self.reads.clear()
    function try_commit(self: &HTx) → bool
8       if ¬validate(self) then rollback(self); return false
9       self.coros.each(|c| c.resume())
10      self.status = Inactive; self.smr.exit(); self.coros.clear(); self.reads.clear()
11      return true
    function unwrap⟨V⟩(self: &HTx, ch: handle ⟨V⟩) → V
12      if ch.is_yield then self.coros.push(ch)
13      return ch.value()
```

validation will still detect the conflict. Note that we could use the value to immediately detect conflicts, but a simple increment suffices, by deferring detection to the next call to `validate`.

4.3.5 Transaction-Level Events

Lastly, Listing 10 presents five functions used for controlling transactions (`begin`, `validate`, `rollback`, `try_commit`, and `unwrap`). Unlike the six functions from Listing 9, which are used by TDS designers, these functions are employed by users of Harmony TDSs.

The `begin` operation marks the thread's transaction as having begun and then opens a dynamic SMR scope, to indicate that this thread is beginning a code region where optimistic accesses to memory will be made. As a transaction executes operations on data structures, it expects each to execute atomically and preserve the data structure's semantics. However, our design does not provide transaction-

level opacity [41]: Instead, transaction T_d can continue performing operations on data structures even after some other transaction's behavior has invalidated T_d. If programmers require stronger guarantees, they can use `validate` to check the transaction's validity *between* data structure operations. Calls to `validate` check the current transaction's read set, which holds all previously observed version numbers. Should any fail to match, the programmer can invoke `rollback` to execute data structure-defined actions (coroutines, Section 4.5) that undo its operations and release ownership of corresponding `HMeta` objects. Note that Harmony does not specify how this rollback is done, nor does it manage program data. Based on the assumption that rollbacks should be rare, it is thus advantageous to only validate (and risk whole-transaction rollback) at fixed points in program code, because this avoids the need for thread checkpointing (e.g., `setjmp`).

The `try_commit` operation implements a basic optimistic commit protocol [149]. Harmony takes ownership of `HMeta` objects eagerly, so it need only `validate`, to check that all previously-encountered versions have not changed. If any have, it rolls back its updates to data structures. Otherwise, it "rolls forward" any data structure clean-up operations, which eventually releases ownership. Note that failed transactions do not automatically retry: it is the programmer's responsibility to restart. Since a programmer can also explicitly call `validate` and roll back upon failure, we ensure that every control flow has exactly one call to `tx.smr.exit`.

Lastly, we provide an `unwrap` function, which is a convenience method for dealing with the cumbersome syntax of C++ coroutines. Briefly, when an operation returns, `unwrap` puts the corresponding coroutine handle into `Tx.coros`, and then extracts the return value. This feature is discussed in more detail in Section 4.5.

4.4 Handling Non-Existence

Prior work can struggle to express the non-existence of a datum or range efficiently. In this section, we briefly explore two challenges that prior work faces when dealing with non-existence of elements in a set, and describe how Harmony addresses each.

4.4.1 Eager Removal Without False Negatives

The removal of nodes must not lead to false negatives during conflict detection. Suppose transaction T_1 wishes to remove node N_a holding value V_a. If it takes ownership of N_a and *ulinks* it, then how would concurrent transaction T_2, wishing to perform an insertion of V_a, know that it can succeed only if T_1 commits first? Certainly T_1 should not take ownership of the predecessor of N_a, as that could result in false conflicts.

We address this problem via the `removing` field. While T_1 *eagerly* takes ownership of N_a and increments `semantic`, it does not unlink N_a; instead it sets `removing`. Per the conditions in Section 4.3.3, any operation whose correctness depends on the contents of N_a must either take ownership or track the semantic version. Thus they cannot operate on N_a, since it is owned. Meanwhile, as the owner of N_a, T_1 can use N_a so as long as operations are aware of the `removing` field. In particular, T_1 can re-insert V_a by (1) logging state of N_a, (2) clearing `removing`, and (3) updating N_a. At commit time, if `removing` is set, N_a will be unstitched.

4.4.2 Avoiding False Positives

In systems like Boosting, it is straightforward to express that a `lookup` observed that 7 was not in a collection that contains 5 and 9, by taking an abstract (read)

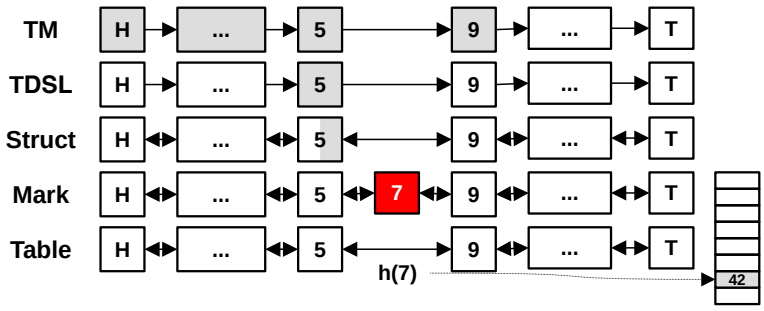


Figure 4.1: Unsuccessful `list::get(7)`.

lock on the value 7. However, it is not clear how a range query could express that all floats between 1.0 and 2.0 are not present, as this would seem to require acquiring abstract locks for every value in the range, to prevent concurrent insertions. In systems like TDSL, the range query condition is easy to express, via the fact that the node holding 1.0 points to the node holding 2.0. However, structure is the only way that TDSL can represent that 7 is not present in a sorted list containing the values 5 and 9, so if another transaction inserts 6, there is a false conflict because $5 \rightarrow 9$ no longer holds. (Of course, systems like TDSL greatly improve upon pure TM, where any change in the prefix of the list from $\perp \ldots 9$ would lead to a false conflict.)

Figure 4.1 depicts the behavior of TM and TDSL, as well as three approaches that are supported by Harmony. The row labeled "Struct" resembles TDSL. That is, it leverages the structural fact that 5 points to 9. `HMeta` has a special field, `structural`, for this purpose. The removal of 9, and any insertion of a value between 5 and 9, will increment the `structural` version of the node holding 5. However, other operations only `log_structural` if they have chosen to use the structural technique to track non-existence. By separating `semantic` and `structural`, some false conflicts common in TDSL can be avoided. We express this by shading *half*

of the node holding 5.

The second technique ("Mark") explicitly inserts a node holding 5, and marks its `removing` field. This effectively makes the absence of 5 *visible* to all concurrent operations, but creates a burden to remove the node when the transaction commits *or aborts*. This is risky, since it can transform a fast, read-only lookup operation into a writing operation. However, it also provides a useful hook for better conflict resolution, since the operation that depends on 5's nonexistence is visible to an operation that might try to insert 5.

Our third technique ("Table") uses the global `notexist` table, which is effectively a table of atomic integers. To use `notexist`, a transaction would hash 5 to a position in `notexist` and log the value, atomically with discovering that 5 is not present. Any insertion of 5 would need to increment that value, so that a subsequent validation would detect the insertion. Note that false positives are still possible, depending on the hash function. In workloads with many data structures that have the same key range, the hash should incorporate a unique key for each data structure, such as its address, so that inserting 5 in one collection does not invalidate failed lookups of that key in another.

All three of these techniques are valuable. For a failed `get`, `remove`, or `insert`, or `update`, we use `Table`. For a mutating range query, we use `Struct` and `Mark` *within the same range query*. When the first or last element in the range is not present, we insert a `Mark` node to represent that fact (if the entire range is empty, we insert two consecutive nodes). By using these `Mark` nodes instead of the `Struct` technique, we avoid the false conflicts that would arise upon a concurrent insertion or removal outside of the range, but immediately preceding (or following) the first (or last) present node of the range. When two keys K_i and K_j are both within the

range, but the keys between them are not present, we use the `Struct` technique, by recording the `structural` version of the node holding K_i. This representation of missing nodes $\{K_i + 1 \ldots K_j - 1\}$ is more efficient than using (potentially many) `Mark` nodes, especially when the universe of keys is not easy to enumerate (e.g., strings, floats). Note that if a data structure chooses to use the `Struct` technique in its range query implementation, then any time a node is inserted or unlinked, the data structure must be sure to increment `structural` in all neighbors of the node. For *read-only* range queries, we use the `Struct` technique even for boundary keys that are absent, so that concurrent, overlapping read-only queries can proceed in parallel.

4.5 Efficient Cleanup with Coroutines

A TDS system must manage the *shape* of the data structure, not just its contents. In some situations, it makes the most sense to defer shape management until a transaction commits or aborts. Consider the following two examples:

List Removal: How should a transaction T remove a value v from a linked list? If it unlinks the node n holding v, then conflicts with a concurrent transaction inserting that same value will be hard to detect, because the `HMeta` associated with the removed node n will not be reachable by the inserting transaction. Furthermore, if T aborts, would it need to re-insert v? If T repeatedly inserts and removes the same v, then if it aborts would it repeatedly remove and insert v during clean-up? If instead T simply takes ownership of the node but does not remove it, then after it commits, how can the transactional system know what to do to physically remove and reclaim the node?

Tree Rebalancing: If T inserts or removes a tree node, and doing so necessitates rebalancing, what should it do? Immediate rebalancing would be "wasted work" if T ultimately aborts. However, if the rebalancing is delayed until T commits, then how much information does the `HTx` need to store in order to rebalance at commit time?

Prior solutions fall into three categories. In Boosting and TM, the structural changes are done immediately; in TM, they increase the conflict set of the transaction. In the TDSL and STO, changes are deferred until commit time, and performed on program data by the TDSL library itself. In Medley, the operation registers a lambda function to run after the transaction commits or aborts.

To make clean-up efficient, we begin by observing that Harmony requires each data structure to use a synchronization mechanism that supports efficient multiword atomic updates (i.e., `S` in Section 4.3.1). This creates an opportunity, because every such `S` supports creating concurrent data structures that have additional inter-node links (e.g., doubly-linked lists instead of singly-linked lists; tree nodes with parent pointers). In such data structures, the only information needed at commit time in order to remove owned node n is a reference to n itself: Its neighbors can be recomputed in $O(1)$ time from these extra links.

Second, we note that removing owned node n is likely to require detailed knowledge of the associated data structure. In a doubly-linked list, removal requires changing two nodes; in an unbalanced binary tree, removal requires changing up to three nodes; in a skip list or balanced tree it may require changing $O(lg(n))$ nodes. Among prior work, Medley offers the most appealing approach, since the data structure designer provides the clean-up code, leveraging data structure-specific knowledge.

Third, we observed above that we do not want cleanup after repeated insertions/removals within a transaction to cause values to be repeatedly inserted and removed during cleanup. Such flickering occurs because individual method invocations lack knowledge of the entire set of operations that a transaction performed. Put another way, operation O's request to run some code to clean up the data structure can be invalidated by a subsequent operation (in the case of commits), or by a preceding operation (in the case of aborts).

To address these challenges, Harmony adds three fields to `HMeta`, and requires each TDS to represent data structure operations as coroutines. This allows the TDS designer to store enough metadata to prevent flicker, without exposing TDS internals to Harmony.

4.5.1 Operations as Coroutines

Coroutines are a generalization of the concept of a procedure [150], which allows a procedure to suspend its execution and later resume.

When the execution resumes, it should pick up from the point of the suspension. It is easiest to think that when a coroutine `co_yields`, it returns a value and a *coroutine handle* to the caller. The caller can later `resume` the coroutine via the handle. There is no limit on the number of times that a coroutine yields.

Since coroutines' stack frames must be stored on the heap, a coroutine's last resumption should `co_return` to provide the caller with a value and inform the run-time system to reclaim the stack frame.

For orthogonality, in Harmony every TDS method is implemented as a coroutine instead of a normal function.

We use coroutines in a limited way: Some only `co_return`, while the rest will

only `co_yield` *once*. This provides a convenient technique for managing data structure cleanup upon commit or abort. When a method is invoked, it computes a result. If no cleanup will be required in the future (e.g., a read-only operation to find a key/value pair), the result is returned; otherwise, the result is yielded, along with a handle. When the result is yielded, the handle is stored in the caller's `HTx.coros` vector; implicitly, the local variables and execution context remain available through that handle.

4.5.2 Linearization of Cleanup With Coroutines

Upon commit or abort, every coroutine in `HTx.coros` is resumed; the order is from oldest to newest in the case of commit (Listing 10, line 9), and from newest to oldest in the case of aborts (line 6). Each resumed coroutine will query the caller's `HTx` to know if the transaction was successful, and then perform the appropriate clean up from the operation attempt. Cleanup uses the synchronization mechanism `S`, along with relevant `HMeta` fields and local (coroutine) state that was computed prior to the `co_yield`. Finally, it uses `co_return` to signal its completion to the TDS system.

Using coroutines in this manner requires a change in how one thinks about linearizability, because coroutines do not have a single invocation and response. Instead they have invocation, `co_yield`, resume, and `co_return` events. We thus say that the coroutine has two linearization points: one between the invocation and `co_yield`, another between the resume and `co_return`. The first is best thought of as being equivalent to the linearization point of an equivalent (non-transactional) data structure operation (e.g., marking a node as deleted), whereas the second can be thought of as reflecting a clean-up operation (e.g., unlinking and reclaiming) that

might, in a non-transactional data structure, be deferred to a later operation. This is sound, because there is a logical transformation of the coroutine into two distinct methods, such that the second is passed an implicit context object produced by the first.

In summary: during transaction execution, coroutines compute results via data structure methods, using ownership and the `removing` field to insert and logically remove nodes without making them visible to other transactions. Clean-up coroutines, which run when the transaction completes, linearize as operations over the structure of the underlying TDSs, removing nodes that are no longer logically present and cleaning up fields of `HMeta`.

4.5.3 Efficient Clean-Up

The above discussion glossed over a key point: An individual cleanup coroutine, on its own, does not konw if there are other cleanup coroutines that need to run on the same node. It is for this reason that we add the `num_coros` and `undo_logged` fields to `HMeta`.

To appreciate their roles, it is helpful to consider a more complex TDS API than has been considered in prior work. Assume that a TDS maps from immutable keys to complex value objects, and that the TDS supports `insert`, `lookup`, `remove`, and `update` operations. If a value is updated, and then the calling transaction aborts, the original value needs to be restored. If a value is logically removed, by setting `HMeta.removing`, and then a new mapping is inserted by the same thread, it cannot simply clear `removing` and write a new value, because the original value must be restored upon abort. Furthermore, in this case, upon commit, the `remove` operation's coroutine should not unlink the node, even temporarily.

We observe there are at most three coroutines that might run. If the transaction commits, then the newest coroutine knows the final state of the node, and should use a linearizable operation, synchronized via S, to finalize that state and release ownership. If the transaction aborts, then the oldest coroutine knows the final state of the node, and should use a linearizable operation, synchronized via S, to finalize that state and release ownership. If the transaction aborts, then the oldest coroutine to change the program data in a node is responsible for restoring the program data to its original state.

To achieve these behaviors, we require every operation that `yields` to increment `HMeta.num_coros`, and also record the updated value of `HMeta.num_coros` on its local stack. We also require every operation that might overwrite the program data in node n to first check the `HMeta.undo_logged` field. If it is `false`, the operation must save the current program data from n onto the stack and set `HMeta.undo_logged` to `true`. (Note that when the oldest coroutine is an `insert`, it can set `undo_logged` to `true`, so that a subsequent `update` by the same transaction does not perform unnecessary undo logging).

Since important information is stored on the coroutine's stack frame, `HTx` does not need to keep any special information, only a list of coroutine handles to resume in forward or reverse order, depending on whether the transaction commits or aborts. Upon commit or abort, each coroutine can tell, using its local state, if it is the newest (oldest), and thus obliged to finalize for commit (abort). This ensures that nodes are never unlinked temporarily, and that ownership is never released too soon, because (1) only one coroutine will modify the node's structure, (2) undo always completes before unlinking, and (3) only the last coroutine will reset fields of the node's `HMeta`.

4.6 Correctness

We argue that Harmony provides correct composition by demonstrating that the fields of `HMeta` can capture the same commutativity conditions as Conflict Abstractions [136]. We use the following set notation: Let $O = \{O_0 \dots O_n\}$ be the set of operations that can be performed on all of the transactional data structures in a program. Without loss of generality, assume that each takes a single argument drawn from $A = \{A_0 \dots A_\infty\}$. Then $O_i(k)$ represents an execution of O_i with argument $k \in A$. We further say that O returns a single value. We use `conflict` as a distinguished return value, indicating that O was not able to complete due to a conflict with some concurrent operation.

While O is running, it is able to read and write the fields of `HMeta` objects and entries in `notexist` (`fields`). These accesses are protected by the same synchronization mechanism that protects the data structure itself, so there is no concern about synchronization. We need only show that the accesses to `HMeta` and `notexist` ensure that an operation is guaranteed to abort if it is concurrent with a non-commutative operation that commits.

When O reads and writes fields of `HMeta`, we can imagine it populates two abstract sets:

- $O_i^r(k)$ contains references to `fields` that $O_i(k)$ read.
- $O_i^w(k)$ contains references to `fields` that $O_i(k)$ wrote.

Given any two operations $O_i(k)$ and $O_j(k')$, the programmer must ensure that if $O_i(k)$ and $O_j(k')$ *do not commute*, and they are performed concurrently by transactions from different threads, then one of the following holds:

- $O_i^r(k) \bigcap O_j^w(k') \neq \emptyset$ (write-read conflict)
- $O_i^w(k) \bigcap O_j^r(k') \neq \emptyset$ (read-write conflict)

- $O_i^w(k) \bigcap O_j^w(k') \neq \emptyset$ (write-write conflict)

Note that $O_i(k)$ need not access and record the `HMeta` fields associated with every node it accesses. It need only access enough fields to construct a sufficient set for conflict detection.

In our design, conflict detection is eager and optimistic: If O_i will make changes to the logical state of some data, it will first transition the corresponding `HMeta.owner` field from unowned to its thread's ID, and will immediately increment the `HMeta.semantic` version. The use of `HMeta.structural` and `notexist` are optional, but the reasoning is the same. This provides an efficient way of detecting many conflicts. Suppose O_i and O_j, issued by concurrent transactions T_i and T_j, conflict. If O_j precedes O_i, but its transaction has not yet committed, then O_i can *eagerly* detect read-after-write and write-write conflicts with O_j while constructing its sets, by observing nodes whose `owned` field is set to T_i. In such cases, it can immediately fail, returning `conflict`. In order to detect write-after-read conflicts, O_i must `log` a `field` value that O_j subsequently increments. Then O_i can detect the conflict via a call to `validate`. Such calls can be explicit within the transaction body, and also are guaranteed to occur at commit time.

4.7 Implementing Data Structures

In this section, we briefly sketch the design of several fundamental data structures. The full Harmony implementation, including these and other data structures, is available at **(redacted for double-blind review)**.

4.7.1 Doubly-Linked List and Ordered Map (Skip List)

Several of our data structures are built atop an ordered map, implemented as a doubly linked list (dlist). This dlist uses head and tail sentinel nodes [108], each of which has an `HMeta` object embedded in it. Each node in the list also has its own `HMeta` object, as well as space for a datum. Regarding keys that are not found, the dlist uses all three techniques from Section 4.4.2. That is, it uses the "Struct" technique for read-only range queries, the "Table" approach when a insert/lookup/remove/update does not find the target key, and the "Mark" and "Struct" techniques for mutating range queries. This means that insertions and removals must increment `HMeta.semantic` when modifying a forward link, and insertions must also update the `notexist` table. We also provide a doubly-linked skip list that extends the dlist's nodes with a variable-sized "tower" containing pointers to predecessors and successors.

By virtue of double-linking, the dlist and skip list can use coroutines for efficient clean-up. This defers unlinking a successful removal until after the operation commits. At that point, the resumed coroutine atomically unlinks the node from the dlist (or skiplist). In order to ensure asymptotic performance when a skip list is accessed several times by a single transaction, insertions eagerly link tall nodes at all appropriate levels. Insertion of a previously-removed node retains the old node's height.

4.7.2 Deque, Stack, and Queue

We also provide a deque based on the dlist, by adding two new fields, the effective head (EH) and the effective tail (ET). Figure 4.2 shows the resulting structure.

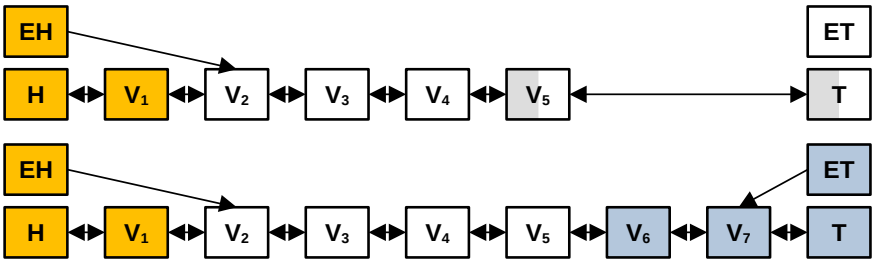


Figure 4.2: Transactional deque with pending `pop_front` *and* `back` *operations (top), and pending* `pop_front` *and* `push_back` *(bottom).*

The deque associates one `HMeta` field with each node. Operations on the front and back can proceed concurrently, unless the deque is empty or only has one element in it. EH and ET are only used by the transaction that owns the head or tail, respectively, and they serve to identify the point at which the owner's next `push` and or `pop` ought to take place in $O(1)$ time. To detect conflicts when different pending transactions own the head and tail, their combined efforts have caused the deque to become empty, and one is attempting to `pop`, we also use ownership of the individual nodes of the deque. In the figure, this ownership is illustrated by a `pop_front` owning both the head and node holding V_1, and a `push_back` owning the tail and the nodes holding V_6 and V_7. `pop` operations use the `removing` field; when a push follows a pop, it can re-use the node, and update EH or ET accordingly. Upon commit, EH (ET) is used to update `head` (`tail`) in constant time. Upon abort, n coroutines may be needed to restore or unlink one element each, in the case that n elements were pushed or popped, each in $O(1)$ time.

4.7.3 Resizable Array (Vector)

Our resizable vector obeys the requirement in C++ that elements are stored contiguously. Thus we provide a separate table of `HMeta` objects for each vector instance. This is, in effect, an ownership record (orec) table from STM [138, 42, 65].

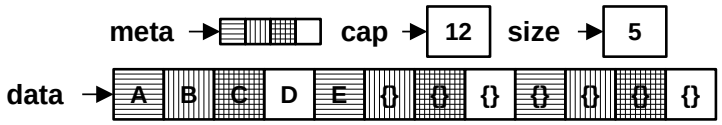


Figure 4.3: Transactional vector. Shading correlates data indices with `HMeta` *objects.*

The programmer specifies the size of the table at construction time. We provide a variant where indices map to `HMeta` objects in a round-robin fashion, as well as one where sets of contiguous indices map to the same object. Figure 4.3 depicts the transactional vector, which consists of an array of elements, a table of `HMeta` objects, and size and capacity fields. For simplicity, this discussion only considers expansion.

An access to entry `i` should not conflict with a `push_back` or `pop_back` when the size is larger than `i`. To this end, `push_back` and `pop_back` take ownership of `size`. In the common case, accessing an element of the vector *reads* the `size` field without checking if it is owned, or recording its semantic version. If `i` is a valid position, then the operation will check if the orec corresponding to `data[i]` is owned. If `data[i]` has actually been pushed or popped by a concurrent transaction, a conflict will manifest on the `HMeta` object, not on the size. If `i` is out of range, the transaction will check if `size` is owned, and abort if it is.

When `push_back` discovers that `size` equals `cap`, resizing is needed. Resizing is a structural operation, but must be coordinated with concurrent accesses to the array elements. Those accesses may be by the same transaction, or conflicting transactions, and any of these transactions can subsequently abort and undo. Our solution is to require any coroutine that will run on account of an abort to record the *index* of the position it modified, not the *address*. Thus an operation can resize the table (and move *owned* data) without taking ownership of anything other than

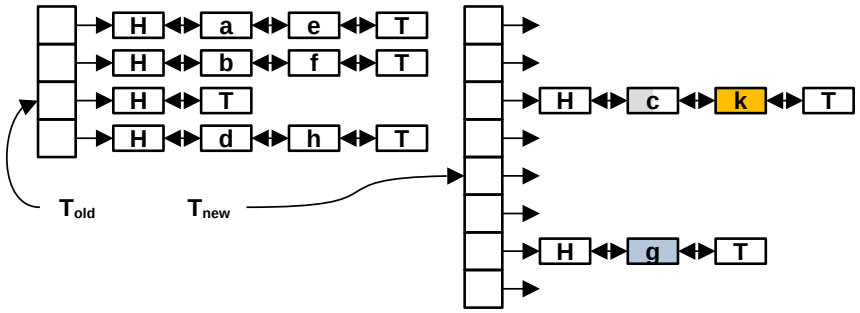


Figure 4.4: Transactional unordered map, implemented as a closed addressing hash table. The third bucket has been resized, without affecting concurrent `insert(k)`*,* `remove(g)`*, or* `get(c)`*.*

the `cap` field itself. Furthermore, if resize is due to `push_back`, it need not undo the resize upon abort: it can even leave the `cap` at its new value.[3]

4.7.4 Unordered Sets and Maps

We also provide a closed addressing hash table, suitable for sets and maps. Figure 4.4 depicts the data structure design, which is based on [119]. The data structure is an array of dlists, where all inserts are at the head of the list. Elemental operations manage ownership of individual nodes in the appropriate list. `Foreach` operations subscribe to each list's head sentinel's structural version, so they can detect concurrent insertions. Read-only `foreach` also subscribe to the semantic version of each encountered node, while mutating `foreach` take ownership of each encountered node.

By carefully managing structural versioning, resizing does not cause self-aborts with preceding `foreach` operations. Inasmuch as resizing is an internal operation, caused by an `insert`, resizing is free to conflict with concurrent `foreach` operations

[3]This design may not be valid in programs that depend on the result of explicit calls to a `capacity` method.

in other transactions. Furthermore, since resizing is structural, it can move nodes that are owned, or whose `semantic` version is being watched, without causing aborts. Note that upon abort, we need not undo the resize.

4.8 Performance Evaluation

In this section, we evaluate Harmony's performance. The current state of the art is Medley, which recently established itself as having the best performance among published TDS systems [133]. When appropriate, we also include results for LFTT [132], OneFile [144], and TDSL [13]. Our test machine has four Intel Xeon Platinum 8160 CPUs at 2.10GHz (96 cores/192 threads) and 754GB of RAM. It runs Ubuntu 22.04.2 LTS with Linux kernel 5.19.0-43-generic. All code was compiled using clang++ 15.0.7 with O3 optimizations, and all data points are the average of five 10-second trials, using the jemalloc allocator [151]. We did not observe significant variance for any of the experiments.

We use the resizable hash table from Section 4.7.4 as our unordered map. Our ordered map is the skip list from Section 4.7.1. Medley's unordered map is based on Michael's lock-free hash table [152], and is not resizable. We used the recommended settings (1M buckets). Medley's ordered map is derived from Fraser's nonblocking skip list [60]. It does not support range queries. Note, too, that Medley's default configuration uses the ralloc persistent allocator [153]. We replaced this with jemalloc, which led to an improvement of more than 10%. We also configured OneFile to use jemalloc.

We implemented two versions of our system. One uses an eager STM for data structure synchronization. The other uses optimistic locking [19, 128]. The optimistic locking technique results in data structures similar to the Heller list [108],

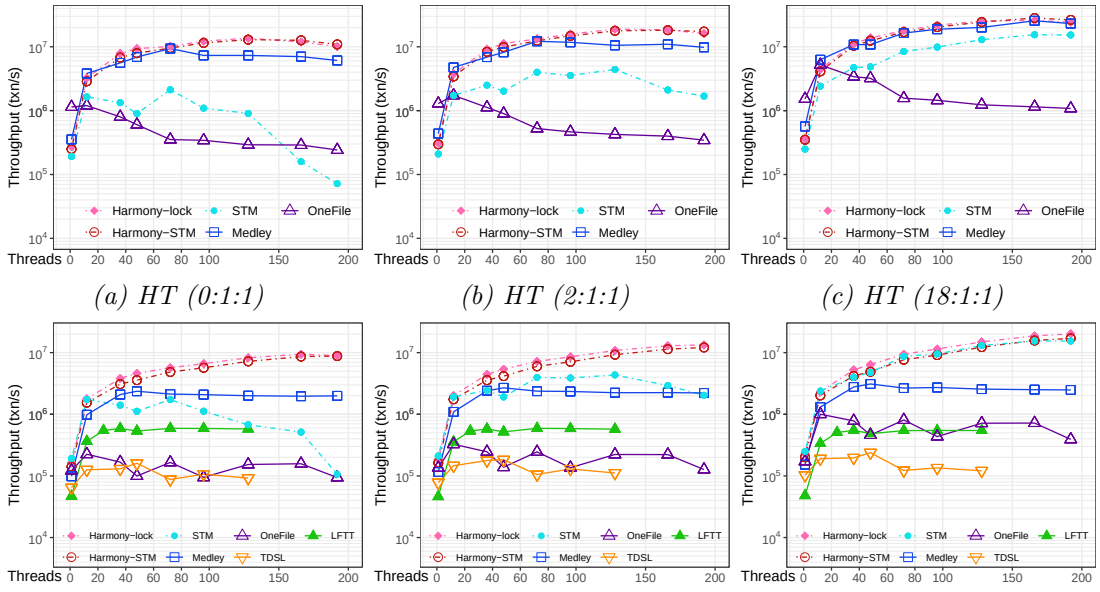
in that they do not acquire locks to read, but acquire locks (and perform lightweight validation) when writing. We also re-implemented each benchmark using the STM from [19]. Among the STM algorithms in that framework, encounter-time locking with undo logging was the best performer.

Since prior work only supports ordered/unordered maps, there is no baseline or standard workload for evaluating the other data structures from Section 4.7. However, rigorous unit tests were developed to verify the correctness of these other implementations, even though they are not evaluated in this chapter.

4.8.1 Microbenchmark Performance

We first look at performance in targeted microbenchmarks. To perform these experiments, we integrated our system into Medley's microbenchmark framework. The throughput test preloads the structure with 0.5M key-value pairs, drawn from a key space of 1M keys. Keys and values are 8-byte integers. In the benchmarking phase, each thread composes and executes transactions of 1 to 10 operations each, with each transaction size equally likely. The operation mix among get/insert/remove is specified as 0:1:1, 2:1:1, or 18:1:1, which ensures the data structure stays at a steady size. Keys are chosen with uniform probability. Results for the hash table and skip list appear in Figure 4.5.

Note that since the Medley implementation of TDSL and LFTT did not include a hash table, we were not able to perform such a comparison. Nonetheless, the relationship between Medley, OneFile, LFTT, and TDSL is roughly the same as reported by Cai et al [133]. In particular, Medley shows the best scalability among prior work. It also outperforms STM in most cases, since writing transactions



(a) HT (0:1:1) *(b) HT (2:1:1)* *(c) HT (18:1:1)*

(d) SL (0:1:1) *(e) SL (2:1:1)* *(f) SL (18:1:1)*

Figure 4.5: Hash table (HT) and skip list (SL) microbenchmark with varied get:insert:remove ratios.

often must validate at commit time, increasing latency. However, the overheads of STM are much lower, confirming recent findings [154, 19] that modern STM is competitive with nonblocking data structures (such as those in Medley).

Harmony matches or exceeds Medley in all experiments. There are a number of sources of improvement. First, our lock-based implementations have low latency, particularly since Medley superimposes a multiword compare-and-swap on top of data structure operations. Second, in our system, operations benefit from the ability to access semantically owned nodes in order to reach their target. This explains the difference in performance and scaling versus STM. While read-only traversals are not wait-free, measurements revealed that very few data structure operations aborted and retried, even when their enclosing-level transactions ultimately failed. Finally, in this work, we did not require opacity, so the validation overhead for our design was minimal, limited to checking a few versions when transactions

committed.[4]

4.8.2 TPC-C Performance

Next, we consider performance on the TPC-C bechmark suite. TPC-C supports five transaction types. Unfortunately, Medley (and indeed all prior work that has been ported to Medley's benchmark system) does not yet support range operations, so they can only execute two transaction types: NewOrder and Payment. We split our evaluation into two parts: First, we consider only these two transactions, so that we can compare our design against prior work. Then we look at how our system performs for the other transaction types.

Figure 4.6 uses the configuration from Medley to run an equal mix of NewOrder and Payment transactions. For low contention, there are as many warehouses as threads, and a static thread:warehouse assignment. For medium contention, we keep the same number of warehouses, but threads pick a warehouse at random for each transaction. We also added a "high contention" workload, where all threads contend on a single warehouse.

TDSL and OneFile performed poorly, with both timing out for many experiments. Thus we focus on the performance of Medley, STM, and our design. Under low and medium contention, Harmony and STM are best. However, the STM curve shows surprising resilience in the face of higher contention. This appears to be a weakness of TDS approaches in general, as Medley (which is obstruction-free) suffers livelock. Since the STM has a built-in contention manager [71], we also include a curve with contention management. This improves performance somewhat, but remains an opportunity for future work. We suspect that the differences between TDS and

[4]Note that Medley is also not opaque.

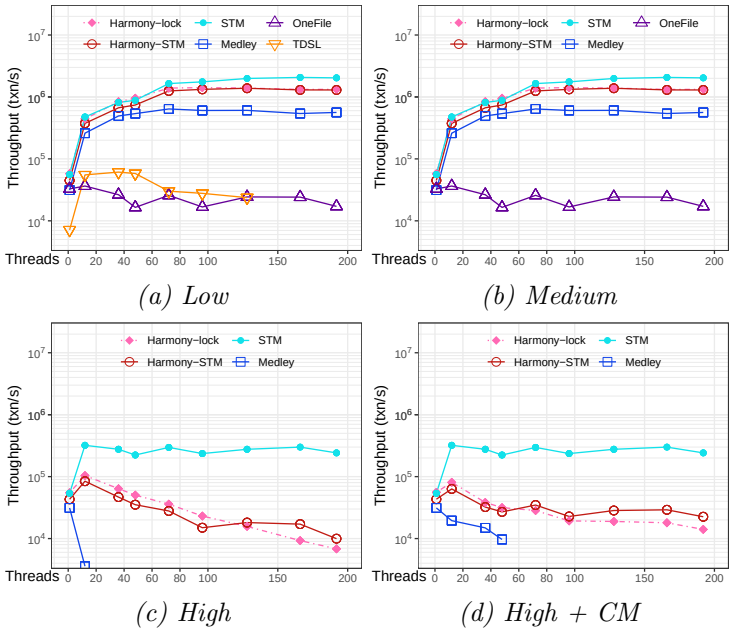
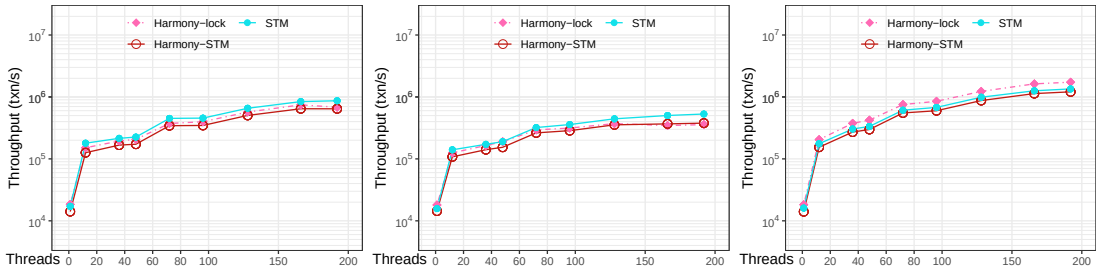


Figure 4.6: TPC-C, 1:1 mix of NewOrder and Payment transactions, parameterized by contention level.

STM may present different challenges and require different solutions for contention management.

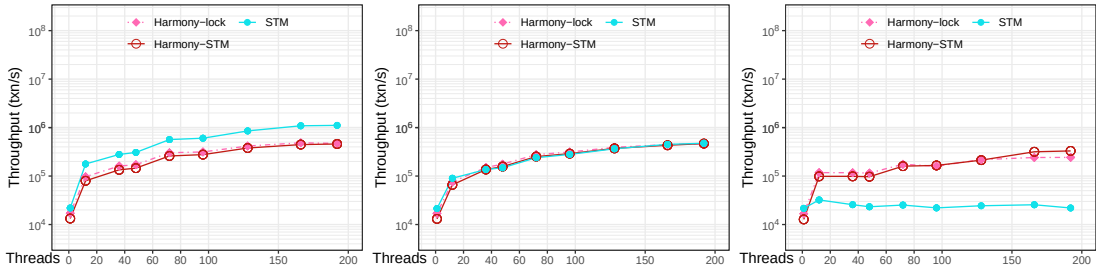
While Medley's skip list does not support range operations, we had no trouble implementing mutating and read-only ranges for our skip list. Figure 4.7 shows the performance of STM and Harmony for an equal mix of OrderStatus and StockLevel transactions, both of which require read-only range operations. Both systems show good scaling. We see a slight advantage for STM, because read-only operations in STM do not need to validate at commit time. However, as contention increases, this benefit diminishes, and at high contention, the systems perform equally well.

Finally, Figure 4.8 shows the performance of a full TPC-C run, using the recommended workload mix of 45% NewOrder, 43% Payment, 4% OrderStatus, 4% Delivery, and 4% StockLevel transactions. Under low contention, both systems



(a) Low contention *(b) Medium contention* *(c) High contention*

Figure 4.7: TPC-C, read-only and range transactions.



(a) Low *(b) Medium* *(c) High*

Figure 4.8: TPC-C, default workload mix, all transaction types, parameterized by contention level.

scale well. As contention increases, STM performance degrades more steeply than Harmony. This differs from Figure 4.6: Now Harmony sustains its performance. The difference stems from STM treating all reads as part of the conflict set, which causes high aborts and flattened scalability. This result suggests that contention management for TDSs deserves further study.

4.9 Conclusions

In this chapter, we presented Harmony, a transactional data structure system that strikes a balance between the two dominant approaches for managing metadata. Harmony separates low-level synchronization metadata from the metadata for

detecting conflicts among transactions. However, its conflict detection metadata does not represent abstract operations, but instead the semantic versions of nodes and values within data structures. While our focus was on metadata, we also introduced new ways of representing non-existence, and showed how coroutines can support efficient cleanup, while keeping management of program data completely outside the scope of the TDS system.

Harmony's design is efficient, leading to low latency and best-in-class scaling. Across standard benchmarks, Harmony outperformed the state of the art, often by a large margin. At the same time, Harmony provides a relatively generic methodology for crafting operations, and thus we were able to implement many data structures, as well as difficult operations such as range queries and hash table resize.

The principles established in this chapter—separating synchronization metadata from transaction metadata, keeping transaction management orthogonal to program data access, and using coroutines for efficient cleanup—form the foundation for extending these ideas to distributed settings. In the next chapter, we explore how the mechanism-policy separation principle can be applied to RDMA-based distributed data structures, where network latency and remote memory access introduce new challenges and opportunities for optimization.

Chapter 5

RDMASTM:Extending Shared-Memory STM to Remote Memory through Layered Policies

Disaggregated architectures expose memory as a network-attached resource over high-speed fabrics like RDMA, enabling flexible resource pooling and failure isolation. However, remote pointer dereference becomes a network round trip, and per-operation costs dominate when spatial locality is not managed explicitly. Building concurrent data structures over RDMA therefore requires careful orchestration of caching, validation, and operation grouping—but existing systems either tightly couple these optimizations with correctness (making reuse difficult) or hide them behind opaque abstractions (sacrificing control).

This chapter presents a layered architecture that separates application/data-structure logic, synchronization semantics, and hardware access. The Access & Placement Substrate (APS) sits between data structures and the synchronization

layer, exposing optimization capabilities—object staging, validation optimization, batching, clock management, and placement control—while hiding hardware-specific details. The synchronization layer invokes APS to realize policies over the hardware fabric. Data structure programmers write against familiar transactional interfaces; optimizations are selected via pluggable policies requiring no algorithm changes. This separation enables systematic exploration of the policy space while preserving STM correctness.

The design is instantiated with lazy STM over RDMA, implementing APS capabilities for four policy dimensions: caching (object-level staging with validation-skip), placement (per-object vs. hashed orecs), clocking (GV1/GV5 with funnel optimization), and batching (segment-aware operation grouping). On CloudLab deployments (1–8 nodes, four data structures, three workloads), evaluation explores how these levers affect performance. Object-level caching delivers the largest gains (nearly 2× throughput, 50–80% RDMA reduction) by collapsing pointer-chasing and skipping redundant validations. Read-set trimming provides orthogonal benefits (16–30% improvement) when data structure semantics permit safe pruning. Clock and batching optimizations show workload-dependent effects: clock variants differ by <20%; batching helps (5–10%) only when memory concentrates and workloads are write-heavy.

These results demonstrate that different policy dimensions have drastically different performance impacts. Caching delivers nearly 2× gains; trimming provides orthogonal benefits; clock and batching effects are workload-dependent. APS provides the right locus for systematic exploration: policies compose without interference, data structure logic remains unchanged, and STM correctness is preserved. While deliberately exploratory (one synchronization layer, one fabric, standard

benchmarks), this work demonstrates that layered separation with pluggable policies enables identifying which optimization levers matter most and should generalize to other synchronization mechanisms (k-CAS, LLX-SCX) and fabrics (CXL, persistent memory).

5.1 Introduction

Disaggregated architectures expose memory as a network-attached resource reachable over high-speed fabrics (e.g., RDMA, CXL, or persistent-memory backends). They are attractive when a system needs more memory capacity, flexible sharing/repurposing across compute nodes, higher effective parallelism, and improved failure isolation and serviceability. This chapter focuses on the case where computation runs on one set of machines while data structures reside—at least in part—in remote memory, and the central challenge is to make the synchronization boundary both programmable and efficient over microsecond-scale fabrics.

Disaggregated architectures—most notably RDMA- and CXL-based designs—embrace heterogeneity: some nodes primarily serve memory while others provide compute; we also include symmetric clusters in which each node can serve both roles, but data are frequently remote over a fast fabric. The dominant cost is the number of network round trips (RTTs); pointer-heavy structures are therefore *RTT-bound*. This pattern is widely observed in prior RDMA systems (e.g., FaRM; HERD; DrTM). We next introduce a layered architecture that addresses RTT/placement directly by separating concerns and exposing the right levers.

Existing systems illustrate two extremes. On one end, co-designed RDMA systems (e.g., FaRM [52], DrTM [155], HERD [156], Sherman [157]) achieve high performance but couple correctness with access mechanics; on the other end, general

RPC abstractions (e.g., eRPC, gRPC/Thrift-style stacks) improve portability but hide fine-grained control and placement. Neither provides a reusable way to *separate correctness semantics from access mechanics* while still exposing the levers that matter for performance. This separation is important: when low-level access is tangled with correctness, caching and placement policies become ad hoc and hard to reuse or reason about; when RPC hides access mechanics, the synchronization layer cannot express the levers that reduce RTTs.

We argue for a *semantic layering* of remote-memory data structures that cleanly separates: (i) application/data-structure logic, (ii) synchronization semantics (e.g., opacity for STM [41], linearizability for multi-word CAS [35, 38]), and (iii) hardware access (RNIC/fabric primitives: one-sided reads/writes, atomics, and fences; local NUMA is a special case of this layer). The key missing piece is an *Access & Placement Substrate (APS)* between the data-structure and synchronization layers. APS sits between (i) and (ii) and provides a narrow, portable interface for *policies*—e.g., object-level staging to collapse pointer chasing, anchoring hot objects, placing ownership records, trimming read sets, and batching by destination—together with the basic access operations needed to realize those policies. The synchronization layer invokes APS, so data-structure programmers write against familiar synchronization mechanisms and opt into hardware-aware optimizations by switching policies rather than rewriting logic.

To make the case concrete, we instantiate one synchronization layer—a lazy, orec-based STM in the style of TL2/ExoTM—and implement APS over RDMA.

Using pointer-based integer sets (lists, hash tables, trees, skip lists), we study ideas that consistently move the needle: object-level staging with a validation-skip rule for cached objects, lightweight anchors for hot data, grouping operations by

destination to reduce doorbells, and a clock-funnel to reduce global-version reads. Crucially, data-structure logic is unchanged as these optimizations are toggled; only the synchronization layer consumes APS capabilities.

Across CloudLab deployments (1–8 nodes; standard integer-set microbenchmarks; uniform key distributions), object-level caching with validation-skip delivers the largest gains (nearly 2× throughput) by collapsing pointer-chasing RTTs; read-set trimming provides orthogonal benefits (16–30%); clock and batching choices show workload-dependent effects (see § 5.5 for detailed measurements). The key finding: different policy dimensions have drastically different performance impacts. These results support APS as the right locus for systematic exploration—enabling identification of which optimization levers matter most for a given workload.

Our goal is architectural: show that APS can be implemented cleanly, and demonstrate that synchronization mechanisms can consume it without entangling hardware details. The evaluation is deliberately narrow (one synchronization layer, one fabric, a modest set of capabilities) and exploratory rather than exhaustive; broader workloads, other synchronization mechanisms, and other fabrics (CXL, PMem) are left for future work.

In brief, this chapter contributes: (1) a layered architecture that separates application logic, synchronization semantics, and hardware access through APS; (2) a working instantiation—an orec-based, lazy STM that maps to APS over RDMA—showing that APS cuts RTTs without changing data-structure code; and (3) evidence distilled into simple practice: manage locality explicitly, place validation to remove round trips, and batch by destination when it helps.

The rest of this chapter proceeds as follows. § 5.2 reviews remote-memory fundamentals and the synchronization semantics we rely on. § 5.3 presents the

programming model and examples. § 5.4 describes APS policies and their implementation. § 5.5 evaluates the capabilities. § 5.6 discusses related work, and § 5.7 concludes.

5.2 Background

Remote Direct Memory Access (RDMA) is a network technology that allows a machine to access preregistered memory on a remote machine directly through its RNIC, keeping the remote CPU off the critical path. [158, 159] RDMA exposes two families of operations: one-sided (READ, WRITE, and 64-bit aatomics such as Compare-and-Swap and Fetch-and-Add), which complete without remote CPU involvement, and two-sided (SEND/RECV), which require a cooperating endpoint. [159, 160] In this work we use one-sided operations exclusively.

Process roles and setup. In the RDMA model, local processes access their own memory via the CPU/memory subsystem, while a remote machine's RNIC exposes preregistered regions for direct access. There is no fixed server/client dichotomy: after an initial handshake that creates queue pairs (QPs) and exchanges memory keys, any node can issue one-sided operations to any other. [159, 160] We assume a stable setup that yields (i) per-peer QPs and (ii) stable remote virtual addresses for registered regions.

Operational essentials used later. We rely only on preregistered, pinned memory with stable remote addresses; one-sided READ/WRITE/CAS; optional batching/inline when multiple requests are pending; and requester-side completion polling. This reflects standard practice for one-sided data-structure access over

RDMA. [158, 52]

Two facts shape data-structure design over RDMA. First, remote pointer chasing expands into multiple network round trips (RTTs) and the cost grows quickly with path length. Second, access granularity matters: fetching contiguous chunks (e.g., whole objects) amortizes RTTs and recovers some of the spatial locality that hardware caches provide in shared memory. [52, 156]

5.2.1 Remus: a thin RDMA substrate

We build APS on *Remus*, a minimal RDMA runtime that abstracts both compute and memory nodes.

A *memory node* is passive: it merely exposes preregistered memory regions that can be remotely accessed by other machines. It does not issue verbs or initiate communication; instead, it acts as a remote RAM pool whose registered regions are accessible via RDMA. All management actions—allocation, reclamation, and metadata updates—are driven by *compute threads* running on compute nodes through one-sided operations. This design keeps memory nodes lightweight, avoiding any CPU involvement in the data path and allowing them to scale as disaggregated memory backends.

A *compute node* hosts one or more *compute threads*, which drive execution by issuing one-sided verbs (READ, WRITE, CAS, FAA) over preregistered regions on memory nodes. Each compute thread can allocate and reclaim memory remotely through Remus's runtime, which maintains allocation tables and free lists stored on memory nodes. All object lifecycle management—allocation, reclamation, and metadata updates—occurs through one-sided operations initiated by compute threads.

Compute threads may be executed by either native threads or stackful coroutines, depending on the desired concurrency and scheduling model. Remus treats them uniformly under a coroutine-compatible interface, enabling seamless overlap of RDMA operations and computation while preserving the one-sided semantics.

Each compute thread maintains a configurable *staging buffer*, a local but RDMA Preregistered region of memory that can be used by user code for arbitrary short-term purposes such as temporary caching, batching, or prefetching. The buffer's size and usage policy are configurable, allowing different applications to adapt Remus to their own access and communication patterns.

Compute and memory roles may coexist on the same physical machine or be deployed separately, allowing compute threads to access disaggregated memory pools transparently.

5.2.2 STM foundations: orecs and global clocks

We adopt a standard orec-based STM that provides opacity and atomic commit. [42, 41] Transactions maintain read and write sets at field or object granularity. Each logical object (which contains one or more fields) is guarded by an ownership record (*orec*) stored in shared memory (here, remote RDMA memory) that holds a version number and a lock/owner indicator. Object-level caching is a per-transaction staging optimization whose correctness follows from orec validation and version-stamped commit; policy details appear in § 5.4.1.

Read protocol. A transaction begins by snapshotting the global timebase G. [43, 61] On first access to an object O's Field F, it read F and then validates $\texttt{orec}(O)$ (post-read validation): if the orec is locked by another writer, the reader aborts;

otherwise, it records the orec's version $v \leq G$ in its read set and proceeds. Because remote memory accesses over RDMA are costly, repeated validations of the same object within a transaction should be avoided. If multiple fields of the same object are accessed, the transaction reads the entire object once and caches it locally. Subsequent accesses to other fields of that object reuse the cached copy and skip further orec validations. This optimization amortizes network round-trip costs while preserving correctness, since the orec's version is still validated once against the snapshot before the object is used.

Write protocol. On the write path, the transaction buffers updates in a redo log (write-back) or performs in-place updates with undo logging (write-through). At commit, it first acquires orecs for all objects in its write set, then validates its read set (orecs unchanged since being read), obtains a commit version from the global clock, writes that commit version into each acquired orec, applies updates (if using write-back), and releases locks. Validating or writing back orecs individually incurs high overhead due to RDMA fences and doorbell signaling. Instead, orecs can be grouped by their destination segment: if multiple targets reside in the same remote region, Remus's `WriteSeq` can coalesce them into a single operation, eliminating these costs. The same optimization applies to writes—buffering field updates locally before flushing them together. Moreover, not all read-set entries require revalidation: depending on the data-structure semantics, users may safely trim the read-set prior to validation, further reducing redundant checks.

Global clock and commit version. The global version clock provides a monotonically increasing timebase that orders commits and orecs validation. [43, 61] We implement two clock families from prior TM literature [43] that preserve snap-

shot semantics while offering different contention profiles: *GV1* (Global Version 1: increment-on-commit via Fetch-and-Add), and *GV5* (Global Version 5: read-plus-one on commit, with livelock avoidance by CAS-advancing the clock on abort). However, this direct implementation can introduce a hotspot bottleneck on the global clock.

We consider two intuitive optimizations. First, a *funnel* can be used to reuse locally fetched copies of the global clock across threads, reducing repeated accesses to the same remote memory location. Second, a *distributed clock* design could eliminate the hotspot entirely by maintaining per-compute thread vector clock and synchronizing them on access failure. We leave this direction for future work, as in our current experimental settings the global clock advances slowly and does not yet form a significant contention point.

Orec placement and mapping. To reduce remote traffic, orecs can be embedded per-object (direct pointer offset, larger footprint, no false sharing) or placed in a shared table indexed by hashing (smaller footprint, potential false sharing). Placement interacts with access patterns (e.g., array-based buckets vs pointer-heavy lists) and with caching strategies (fetching entire objects can turn multiple orec checks into one); see § 5.4.2. [42, 61]

These observations motivate introducing an **Access and placement (APS)** layer that exposes such optimization opportunities to data-structure designers, enabling them to explore placement, caching, and clocking strategies without modifying synchronization or RDMA logic.

Listing 11: Read-only data structure operation (search)

```
   function get(me: Desc, key: K)
1      while true do
2          begin_RO(me)                                  // read-only scope|no locks acquired
3          n ← find_predecessor(me, key)
4          if n = null then continue
                                                         // validation failed, retry
5          nkey ← n → key.get(me, n)
6          if nkey = null then continue
                                                         // validation failed
7          if nkey = key then
8              end_ro(me)                                // validation succeeded, no commit needed
9              return true
10         end_ro(me)
11         return false                                  // key not found
```

5.3 Programming Model

In this section, we describe APS from the programmer's perspective. Conceptually, APS appears as just another *policy* within an existing synchronization mechanism. As a result, an already correct STM-based data structure can be adapted with only minimal modifications to take advantage of APS capabilities.

First, we use a doubly linked list as an example. Listing 11 shows the `get` method, and Listing 12 shows the `insert` method. The STM used here requires the programmer to **explicitly check and restart** transactions upon conflict detection; transactions begin with `begin_ro` or `begin_rw` and end with `end_ro` or `try_end_rw`. This explicit style exposes the underlying transactional boundaries, making it clear where validation, conflict detection, and commit decisions occur.

The `get` operation (Listing 11) illustrates a **read-only transactional pattern**: it repeatedly starts a read-only scope, traverses the list to find the predecessor node, validates the traversal by re-reading the orec versions, and commits if no conflict is detected. Because no updates are performed, the read-only transaction ends immediately after validation succeeds.

In contrast, the `insert` operation (Listing 12) demonstrates a **read-write**

Listing 12: Read-only data structure operation (insert)

```
struct node_t
    FIELD⟨node_t*⟩ prev, next
struct data_t: node_t
    FIELD⟨K⟩ key
    FIELD⟨V⟩ val

function insert(me: Desc, key: K, val: V)
1   while true do
2       begin_RW(me)                                  // begin RW, snapshot global clock
3       n ← find_predecessor(me, key)
4       if n = null then continue
                                                      // validation failed during traversal
5       nkey ← n → key.get(me, n)
6       if nkey = key then
7           try_end_rw(me)                            // key exists|commit read-only path
8           return false
9       next ← n → next.get(me, n)
10      if next = null then continue
                                                                  // validation failed
11      nd ← LOG_NEW⟨data_t, me⟩()                    // allocate in RDMA, visible at commit
12      nd → key.set(me, nd, key)                     // initialize via set (required for FIELD)
13      nd → val.set(me, nd, val)
14      nd → next.set(me, nd, next)                               // link to successor
15      nd → prev.set(me, nd, n)                                  // link to predecessor
16      if ¬n → next.set(me, n, nd) then continue
                                                                  // linking failed, retry
17      if ¬next → prev.set(me, next, nd) then continue
                                                                  // conflict detected
18      if ¬try_end_rw(me) then continue
                                                      // commit failed, full retry
19      return true
```

transactional pattern. It begins by locating the correct predecessor node using the same validation rules, then allocates and initializes a new node within the transaction's redo log. The new node's fields are written through `set` calls, which record pending updates. Linking steps are performed atomically through transactional field updates (e.g., `n→next.set(me, n, nd)`), and any validation failure restarts the entire transaction. A successful `try_end_rw` ensures that all modifications become visible atomically, preserving list consistency across concurrent transactions.

Together, these examples show that **APS does not alter STM semantics**—it merely provides additional optimization policies (for placement, caching, batching, etc.) under the same programming interface. From the programmer's view, adopting

APS means selecting a different policy rather than rewriting the transactional logic itself.

However, there is sevral subtle difference compared to the shared-memory case. In a local STM, a field such as `key` can be declared `const` as long as we know it will never be reset. Under an RDMA policy, however, even immutable fields must be wrapped as `FIELD` objects, because their values reside in remote memory. Therefore, every access must go through `get` to ensure that the value is fetched and validated from the remote location. For the same reason, APS does not expose C++'s `new` or placement-`new` directly on transactional objects; instead, programmers allocate objects through APS's memory manager so that the system can register and map them correctly. Although these constraints currently require explicit calls (get(), APS allocation), many of them could be hidden behind operator overloading or wrapper types if one wished to provide a more idiomatic C++ interface, while still preserving the underlying synchronization and validation semantics. APS also performs the object's field initialization during allocation so that the created object can be published and validated consistently. For the same reason, APS does not support C++'s `new` or placement-`new` operators directly. Instead, programmers allocate transactional objects through `LOG_NEW`, which obtains memory from the APS layer's memory manager and must use explicit `set` for object field's initialization. Depending on the active memory-allocation policy, the underlying segment may reside in remote or local memory. Although not shown in the code, APS also provides a `reclaim(nd)` API for freeing memory allocated through `LOG_NEW`. Direct use of `delete` is restricted because deallocating memory outside APS would bypass orec ownership, violate transactional ordering, or prematurely free remote regions. In practice, APS provides its own deallocation

Listing 13: Initialization procedures for leader and follower

```
   class dlist
       FIELD⟨node_t*⟩ head                                   // list head pointer
       FIELD⟨node_t*⟩ tail                                   // list tail pointer
   function init_lead(me: Desc, params, ds_ptr, aps)
1      begin_RW(me)                                          // begin RW transaction
2      head ← LOG_NEW⟨node_t, me⟩()                          // allocate head in RDMA
3      tail ← LOG_NEW⟨node_t, me⟩()                          // allocate tail in RDMA
4      head → next.set(me, head, tail)
5      tail → prev.set(me, tail, head)
6      try_end_rw(me)
7      head_ptr ← ds_ptr + offset_of(head)
8      aps.write(head_ptr, head)                             // publish head to remote
9      tail_ptr ← ds_ptr + offset_of(tail)
10     aps.write(tail_ptr, tail)                             // publish tail to remote

   function init_follow(params, ds_ptr, aps)
11     head_ptr ← ds_ptr + offset_of(head)
12     head ← aps.read(head_ptr)                             // read published head
13     tail_ptr ← ds_ptr + offset_of(tail)
14     tail ← aps.read(tail_ptr)                             // read published tail
```

routine, and a C++ interface could overload operator `delete` to route deletions safely through APS while preserving idiomatic syntax. It should be mentioned, to simplify object caching and remote access, APS currently assumes that each transactional object has a **fixed size**. Variable-length objects (e.g., those containing flexible array members such as `array[]`) are not yet supported. Supporting variable-length layouts would require augmenting the `get`/`set` interface with an explicit length parameter, allowing APS to determine the object's true size when fetching or updating remote memory. Unlike fixed-size objects, this length information cannot be inferred automatically, making it difficult to hide behind operator overloading or implicit wrappers. As a result, variable-length objects represent one of the few cases where the APS interface must remain explicit.

In order to be executed correctly, each data structure must also provide two initialization hooks—`init_leader` and `init_follow`—which are invoked during application startup. The leader routine (`init_leader`) is responsible for initializing and publishing shared objects in RDMA memory in the example showing in

Listing 14: Application setup: leader/follower initialization and data structure operation execution

```
   // Leader (one per system)
   if is_leader_fiber() then
1      me ← create_STM_context()
2      root ← aps.allocate⟨ds_t⟩ in RDMA
3      ds_ptr.init_lead(me, params, root, aps, use_anchor)
4      publish_root(root)

   // Follower
   else
5      me ← create_STM_context()
6      root ← get_published_root⟨ds_t⟩
7      ds_ptr.init_follow(params, root, aps, use_anchor)

   // Application operations
8  found ← ds_ptr.get(me, key)
9  inserted ← ds_ptr.insert(me, key, value)
```

Listing 13: it allocates and links the initial nodes, commits them transactionally, and writes their addresses to globally visible locations using the APS layer. Follower nodes then invoke `init_follow` to attach to this published state, retrieving the shared pointers via `aps.read` and reconstructing their local views of the data structure. Together, these hooks establish a consistent shared-memory image across all participants before transactional operations begin. Listing 14 shows how these hooks are integrated into the application's setup sequence.

5.4 APS and Policies for RDMA

This section details the internal design of the Access & Placement Substrate (APS) policies, which extend the basic STM API introduced in § 5.3 and provide additional optimization interfaces. Each policy governs a distinct optimization axis—caching, placement, batching, and clock management—while preserving STM correctness. Together, these policies form the foundation that enables the same high-level transactional logic to execute efficiently over RDMA without altering its semantics.

APS wraps the underlying Remus substrate for convenience and exposes basic

Listing 15: Caching policies for RDMA operations (not cache)

```
// No-cache policy: direct RDMA operations
class nocache_t
    function get_address(field, obj, for_write)
1       return field                                    // no translation needed
    function get_value(field)
2       return aps.read(field)                          // direct RDMA read
    function set_value(field, value)
3       aps.write(field, value)                         // direct RDMA write
```

verbs such as read, write, CAS, FAA, allocate, and reclaim. However, these primitives are identical to those provided by Remus itself. Conceptually, the APS layer should be regarded as a collection of *policy layers* for synchronization mechanisms, rather than as an extension of the Remus interface.

5.4.1 Cache Policy and skip validation mechanism

We begin with cache policy:**no-cache policy** is the simplest policy, shown in Listing 15.The *no-cache* configuration issues RDMA verbs through Remus directly for every read or write, mapping object fields to their remote addresses without any local buffering. Each `get_value` performs a one-sided `aps.read`, and each `set_value` a corresponding `aps.write`. This policy serves as the **baseline**, representing pure one-sided access with minimal state but maximal network round-trips. It is correct under STM semantics but incurs the highest latency.

Listing 16 introduces a lightweight write-back cache to reduce repeated remote reads and collapse small writes, we call it **cache policy**. The `cache_t` structure maintains per-object cache blocks, fetched on demand via `ensure_cache`. Fields are then accessed directly through cached local addresses, marking blocks dirty when modified. At commit time, `write_back` pushes updated regions back to RDMA memory using one-sided writes. This design reduces read latency and allows

Listing 16: Caching policies for RDMA operations

```
    struct cache_block_t
        uint8_t* local_addr                                   // local cache address
        uint64_t size                                                 // block size
        bool dirty                                            // has pending writes?

    // Caching policy: maintain local cache with write-back
    class cache_t
        // cache:  map from address to cache blocks
        function ensure_cache(addr, size)
1           curr_size ← 0
2           foreach block ∈ cache[addr] do
3               curr_size ← curr_size + block.size
4               if curr_size ≥ size then return true
                                                                   // cache hit
5           extend ← size − curr_size
6           local_addr ← local_allocate(extend)
7           aps.read(addr + curr_size, local_addr, extend)          // fetch data
8           cache[addr].add(local_addr, extend, false)
9           return false                                            // cache miss
        function get_address(field, obj, for_write)
10          if ¬field_in_object(field, obj) then
                                                          // field outside object?
11              hit ← ensure_cache(field, sizeof(field))
12              cache[field][0].dirty ← cache[field][0].dirty ∨ for_write
13              return (cache[field][0].local_addr, hit)
14          hit ← ensure_cache(obj, sizeof(obj))
15          offset ← field − obj
16          sum ← 0
17          foreach block ∈ cache[obj] do
18              if offset < sum + block.size then
19                  block.dirty ← block.dirty ∨ for_write
20                  return (block.local_addr + offset − sum, hit)
21              sum ← sum + block.size
        function get_value(field)
22          return ∗field                                 // read from local cache
        function set_value(field, value)
23          ∗field ← value                                 // write to local cache
        function write_back()
24          foreach (obj, blocks) ∈ cache do
25              offset ← 0
26              foreach block ∈ blocks do
27                  if block.dirty then
28                      aps.write(obj + offset, block.local_addr, block.size)
29                  offset ← offset + block.size
        function clear()
30          aps.reset_cached_slice()
31          cache.clear()
```

local computation on cached data. It should be mentioned, if the field is not inside object then the cache wouldn't really help, and because we use type instead of real

Listing 17: Lazy field operations for transactional memory

```
   // Lazy field: reads check redo log first, writes buffer in redo log
   class lazy_field_t
       function get(tx, obj)
1          if ¬tx.in_tx then abort                                  // not in transaction
2          // check redo log first
3          (log_hit, cache_hit, ret) ← tx.redolog.get(field, obj)
4          if log_hit then return ret
                                                                    // found in redo log
5          // read from cache
6          ret ← tx.redolog.safe_read(field, obj, tx.cache)
7          if cache_hit then return ret
                                                                  // cache hit, skip validation
8          // validate ownership record
9          orec ← obj.orec()
10         locked ← false
11         if tx.exo.check_orec(orec, locked) = tx.eot then
12             tx.unwind()                                        // validation failed, abort
13             return null
14         tx.readset.add(orec)
15         return ret
       function set(tx, obj, value)
16         if ¬tx.in_tx then abort                                  // not in transaction
17         orec ← obj.orec()
18         tx.lockset.add(orec)                                    // defer locking to commit
19         tx.redolog.insert(field, obj, value)                       // buffer write locally
20         return true                                          // writes never fail (redo-based)
```

object size, so variable array wouldn't really get cached too.

To transparently integrate the APS cache with STM, redo and undo logs used by various STM implementations can be built directly on top of this cache. Given an `rdma_ptr`, the runtime can invoke `get_address` to translate remote addresses and use `get_value` and `set_value` in place of ordinary load and store operations.

STM can further exploit cache through the **skip mechanism** (Listing 17). The cache itself is unaware of transactional semantics, yet STM can infer when cached fields have already been validated—either their own ownership records (orecs) or those of their containing objects. When such validation has occurred within the same transaction, repeated orec checks can be safely elided. In lazy STM (as shown in the `get` path), a cache hit therefore implies that the corresponding version

Listing 18: Ownership record (orec) mapping policies

```
   // Per-Object (PO): embedded orecs in allocator header
   class orec_po_t
       // Global state:  none required
       class ownable_t
           function orec()
1              return this − 8                                  // orec in allocator header

   // Per-memory-Stripe (PS): hashed orec table
   class orec_ps_t
       // Global state:  orec_table and num_orecs_
       struct global_t
           rdma_ptr<orec_t>* orecs_                                          // orec table
           uint64_t num_orecs_                                               // table size
           function get_orec( obj )
2              index ← hash(obj) mod num_orecs_
3              return orecs_[index]
       class ownable_t
           function orec(globals)
4              return globals.get_orec(this)                        // hash-based lookup
```

has already been confirmed valid, allowing STM to skip redundant validation. This lightweight elision reduces read-path latency while preserving opacity and correctness.

5.4.2 Orec mapping Policy and Safe Memory Management

APS provides the mechanisms for memory allocation, reclamation, and orec mapping, while STM utilizes these mechanisms together with orec-based validation to ensure safe memory management.

Remus employs two allocators: a local allocator with size-segregated free lists and a global bump allocator backed by the RDMA memory pool. When a compute thread allocates memory, it first attempts to reuse a block from its local free list; if no suitable block exists, it falls back to the global allocator, which carves new blocks from the remote pool. Each allocated block reserves its first 8 bytes to store its size and the next 8 bytes as a general-purpose slot. APS defines how this slot

Listing 19: Memory management in lazy_t STM policy

```
   // Transaction state: pending allocations and reclamations
   vector mallocs                                  // speculatively allocated objects
   vector frees                                    // objects to reclaim on commit

   // Allocate object speculatively during transaction
   function LOG_NEW<T>(size)
1      if ¬in_tx then abort                                  // not in transaction

2      ptr ← aps.allocate(T, size)                           // allocate via RDMA
3      mallocs.add(ptr)                              // log for cleanup on abort
4      return ptr

   // Schedule object for reclamation on commit
   function reclaim<T>(obj)
5      frees.add(obj)                                // defer deallocation to commit

   // Cleanup on transaction abort
   function unwind()
6      ...
7      foreach ptr ∈ mallocs do
8          aps.deallocate(ptr)                       // free speculative allocations
9      ...

   // Finalize on transaction commit
   function try_commit()
10     ...
11     foreach ptr ∈ frees do
12         aps.deallocate(ptr)                             // reclaim deleted objects
13     ...
14     return true
```

is used: in the per-object (PO) mapping policy (Listing 18), the slot holds an embedded ownership record (orec), whereas in the per-stripe (PS) mapping policy, APS allocates and maintains a global orec table on the memory node and maps objects to entries in this table through hashing. Both mappings are supported by the same Remus allocator hierarchy and selected through configuration.

During reclamation (Listing 19), Remus does not immediately return freed blocks to the global pool. Instead, each block is placed into the local free list of the thread that reclaimed it, ensuring that subsequent reuse incurs no cross-thread communication. This simple, strictly per-thread reuse policy avoids introducing additional RDMA synchronization and is sufficient for the purposes of RDMASTM, since our focus is on the interaction between ownership metadata, caching, and

validation—not on global memory recycling.

Importantly, Remus does not currently implement any mechanism for pushing surplus blocks back to the global pool or balancing free lists across threads during execution. Designing policies for global reclamation, fairness, or producer–consumer balancing would require additional coordination protocols and is an orthogonal problem to the STM and APS mechanisms studied in this chapter. APS simply exposes allocate and reclaim as primitives, and STM layers log allocations speculatively, invoking reclamation only after a transaction commits. A more sophisticated global reclamation strategy could be integrated in future work, but it lies outside the scope of RDMASTM's design goals.

Together, APS and STM jointly guarantee safe memory reuse within RDMASTM. APS ensures that reclaimed memory remains physically valid and thread-local until reused, while STM provides logical safety by enforcing orec-based validation. Suppose transaction T1 commits and reclaims an object or field. Any concurrent transaction T2 that still holds a pointer to that memory must validate the object's orec—either the in-object orec under PO mode or the hashed orec entry under PS mode—before dereferencing it. Because reclamation bumps the corresponding orec version, T2 will detect the mismatch during validation and abort before it can read stale state.

A key property of our design is that reclaimed memory is never unmapped or physically destroyed: APS only recycles addresses, it never makes them inaccessible. Therefore, even if a thread transiently holds a stale pointer, dereferencing it cannot cause a fault; at worst, it will trigger an STM abort through the normal orec-checking discipline. This division of responsibility—APS ensuring physical safety, STM ensuring logical safety—removes the need for complex hazard-pointer, RCU,

Listing 20: Base clock interface for RDMA STM

```
// Base clock: common state and utilities
class rdma_clock_t
    // State:  global clock pointer and abort tracking
    rdma_ptr⟨uint64_t⟩ g_
    uint64_t last_abort_                                    // time of last abort

    // Constants
    const uint64_t END_OF_TIME = ULLONG_MAX                 // end of time
    const uint64_t LOCK_BIT = 1 ≪ 63                        // MSB is lock bit

    function is_locked(val)
1       return val ∧ LOCK_BIT                               // test lock bit

    function is_aborted(val)
2       return val ≠ END_OF_TIME                            // END_OF_TIME means not aborted

    function on_abort(abort_time)
3       last_abort_ ← abort_time                            // save abort timestamp
```

Listing 21: GV1 clock: direct global clock access

```
// GV1: every thread directly accesses global clock via RDMA
class gv1_rdma_clock_t extends rdma_clock_t
    // Inherits:  g_, last_abort_, is_locked(), is_aborted()
    // Uses:  aps (APS context for RDMA operations)

    function get_time()
1       return aps.read(g_)                                 // read global clock

    function on_end()
2       if ¬is_aborted(last_abort_) then
3           // committing:  increment global clock
4           return 1 + aps.FetchAndAdd(g_, 1)
5       // aborting:  reset and retry
6       last_abort_ ← END_OF_TIME
7       return null
```

or epoch-based schemes in our RDMASTM exploration.

5.4.3 Clock and funnel mechanism

The clock is central to STM correctness and performance. Every transaction needs timestamps to establish a consistent snapshot at start and to serialize commits. In RDMA environments, these timestamps come from a global clock stored in remote memory, accessed through one-sided operations. APS provides multiple clock policies that balance synchronization cost against consistency guarantees. All

Listing 22: GV5 clock: commit uses read+1, abort advances clock

```
   // GV5: committers read+1, aborters ensure clock advances
   class gv5_rdma_clock_t extends rdma_clock_t
       // Inherits:  g_, last_abort_, is_locked(), is_aborted()
       // Uses:  aps (APS context for RDMA operations)

       function get_time()
1          return aps.read(g_)                                    // read global clock

       function on_end()
2          if ¬is_aborted(last_abort_) then
3              // committing:  read + 1 (no increment)
4              return aps.read(g_) + 1
5          // aborting:  advance clock past abort time
6          if is_locked(last_abort_) then last_abort_ ← END_OF_TIME    // locked orec, just
            retry
           return null
7          // ensure global clock ≥ last_abort
8          while true do
9              time ← aps.read(g_)
10             if time ≥ last_abort_ then break
11             ret ← aps.CompareAndSwap(g_, time, last_abort_)
12             if ret = time ∨ ret ≥ last_abort_ then break
13         last_abort_ ← END_OF_TIME
14         return null
```

policies share a common base interface (Listing 20), which defines the `rdma_clock_t` structure with shared state—a pointer to the global clock (`g_`) and the timestamp of the most recent abort (`last_abort_`). The base also provides utility functions: `is_locked` tests if a timestamp has its lock bit set, `is_aborted` checks if an abort has occurred, and `on_abort` records the conflicting version that caused an abort. Each derived policy implements two key methods: `get_time` to read the current clock, and `on_end` to finalize the transaction by either returning a commit timestamp or handling abort-induced clock updates.

We begin with the **GV1 clock** (Listing 21), the simplest versioning policy. On transaction start, `get_time` performs a remote read of `g_` to establish the transaction's snapshot time. On commit, `on_end` increments the global clock via `FetchAndAdd` and uses `1 + FetchAndAdd(g_, 1)` as the commit timestamp. This approach guarantees strict serialization: every commit receives a unique,

Listing 23: GV1 with funnel: local clock optimization

```
    // GV1 funnel: local clock to reduce remote reads/writes
    class gv1_rdma_funnel_clock_t extends rdma_clock_t
        // Inherits:  g_, last_abort_, is_locked(), is_aborted()
        // Uses:  aps (APS context for RDMA operations)
        // State:  local funnel clock
        atomic⟨uint64_t⟩* local_clock_                                  // machine-local funnel

        function read_funnel_clock()
1           while true do
2               waited ← false
3               while is_locked(time ← local_clock_) do
4                   waited ← true                                       // spin while locked
5               if waited then return time
                                                                        // peer updated, use it
6               // try to lock and update from global
7               if CAS(local_clock_, time, time ∨ LOCK_BIT) then
8                   break
9           time ← aps.read(g_)                                         // fetch from global
10          store(local_clock_, time)                                   // update local, unlock
11          return time

        function increment_global_clock()
12          first ← −1
13          while true do
14              while is_locked(time ← local_clock_) do
15                  if first = −1 then first ← time
16              if first ≠ −1 ∧ (first ∧ ¬LOCK_BIT) < time then return time
17              if CAS(local_clock_, time, time ∨ LOCK_BIT) then
18                  break
19          time ← aps.FetchAndAdd(g_, 1) + 1
20          store(local_clock_, time)                                   // update local, unlock
21          return time

        function get_time()
22          return read_funnel_clock()

        function on_end()
23          if ¬is_aborted(last_abort_) then return increment_global_clock()
24          last_abort_ ← END_OF_TIME
25          return null
```

monotonically increasing timestamp, and the hardware atomic operation totally orders concurrent commits. The downside is contention—all committing threads compete for the same remote word, creating a serialization bottleneck. Under heavy write workloads, this manifests as increased abort rates and reduced throughput.

The **GV5 clock** (Listing 22) eliminates this bottleneck by removing the `FetchAndAdd`. Committing transactions simply read the global clock and use

Listing 24: GV5 with funnel: local clock for GV5

```
    // GV5 funnel: local clock reduces remote operations
    class gv5_rdma_funnel_clock_t extends rdma_clock_t
        // Inherits:  g_, last_abort_, is_locked(), is_aborted()
        // Uses:  aps (APS context for RDMA operations)
        // State:  local funnel clock
        atomic⟨uint64_t⟩* local_clock_                                  // machine-local funnel

        function read_funnel_clock()
1           while is_locked(time ← local_clock_) do
                                                                                  // spin
2           return time                                                  // local read only

        function read_funnel_clock_remotely()
3           while true do
4               waited ← false
5               while is_locked(time ← local_clock_) do
6                   waited ← true                                        // spin while locked
7               if waited then return time
                                                                         // peer updated, use it
8               if CAS(local_clock_, time, time ∨ LOCK_BIT) then
9                   break
10          time ← aps.read(g_)                                          // fetch from global
11          store(local_clock_, time)                                    // update local, unlock
12          return time

        function ensure_min_global_clock()
13          while true do
14              time ← read_funnel_clock()
15              if time ≥ last_abort_ then last_abort_ ← END_OF_TIME      // already advanced
                return
16              if CAS(local_clock_, time, time ∨ LOCK_BIT) then
17                  break
18          // locked funnel, try greedy CAS on global
19          ret ← aps.CompareAndSwap(g_, time, last_abort_)
20          if ret = time then store(local_clock_, ret + 1)
                                                                                  // CAS succeeded
21          else store(local_clock_, ret)
                                                                 // CAS failed, use new value
22          last_abort_ ← END_OF_TIME

        function get_time()
23          return read_funnel_clock()                                   // local only

        function on_end()
24          if ¬is_aborted(last_abort_) then
25              // committing:  read global + 1
26              return read_funnel_clock_remotely() +1
27          // aborting:  ensure clock advances
28          if ¬is_locked(last_abort_) then ensure_min_global_clock()
29          last_abort_ ← END_OF_TIME
30          return null
```

`read(g_) + 1` as their commit timestamp. Multiple threads can now commit concurrently without serializing through a shared atomic operation. However, this optimization introduces a subtle livelock hazard. Suppose transaction T aborts after observing an orec with version v. If the global clock remains below v (because no other transaction has committed recently), T will read a start time less than v on restart and abort again, repeating indefinitely. To break this cycle, GV5's `on_end` detects when a transaction is aborting (by checking `is_aborted(last_abort_)`) and advances the global clock past `last_abort_` using a CAS loop. There is one exception: if the conflicting orec is currently locked (`is_locked(last_abort_)`), the transaction simply resets and retries without touching the clock, allowing the lock holder to advance it naturally on commit. This trade-off reduces contention at the cost of additional clock-advancement logic on abort.

Both GV1 and GV5 can be enhanced with a **funnel mechanism** (Listings 23 and 24). The funnel is a machine-local clock shared by all threads on the same compute node. It acts as a combining layer to reduce remote operations: threads attempt to acquire the funnel (using a lock bit in its most significant bit), and the first successful thread performs the expensive remote operation and updates the local clock, allowing waiting peers to reuse the result. For GV1-funnel, `increment_global_clock` shows this pattern: threads spin-wait on the local clock while it is locked, and if they observe the clock advance while waiting, they return immediately without a remote operation. Otherwise, they lock the funnel via CAS, perform `FetchAndAdd(g_, 1)`, store the result locally, and release the lock. This coalesces multiple increments into a single remote operation. For GV5-funnel, `read_funnel_clock_remotely` applies the same pattern to reads, while `ensure_min_global_clock` coalesces CAS-based clock advancements during

abort handling. The funnel optimization is most effective when multiple threads commit or abort simultaneously on the same node, reducing network round-trips proportionally to thread count.

Together, APS exposes these clock policies through a uniform interface, allowing STM implementations to select the appropriate trade-off for their workload. At transaction start, STM invokes `get_time` to establish a snapshot timestamp. During validation, when an orec check fails, STM calls `on_abort` to record the conflicting version. At transaction end, STM invokes `on_end`, which either returns a commit timestamp (for committing transactions) or handles clock advancement to prevent livelock (for aborting transactions). This separation allows the STM algorithm to remain clock-agnostic while APS manages the underlying versioning mechanism and livelock prevention.

The funnel pattern generalizes to other hot metadata. APS provides an **anchor** mechanism for frequently-accessed remote objects such as data structure roots, sentinels, and orec tables. Like the funnel, an anchor maintains a thread-local cached copy with version tracking, turning repeated remote reads into local loads with occasional version refresh. When a thread accesses an anchored object, it first checks the local copy's version against a cached timestamp; if valid, the access completes locally without any RDMA operation. On version mismatch or first access, the thread performs a remote read to refresh both the object data and the version. This pattern is particularly effective for metadata that is read frequently but modified rarely, such as the head sentinel in a linked list or the root pointer in a tree.

Listing 25: Caching policies for RDMA operations (batch optimization)

```
   // Batched caching: extends cache_t with batched RDMA writes
   class batch_cache_t extends cache_t
       function write_back()
1          // Group dirty blocks by segment ID
2          foreach (obj, blocks) ∈ cache do
3              offset ← 0
4              foreach block ∈ blocks do
5                  if block.dirty then
6                      s ← aps.seg_id(obj)
7                      group[s].add((obj + offset, block))
8                  offset ← offset + block.size
9          // Write back grouped blocks with batching
10         foreach s ∈ group do
11             foreach i ← 0 to |group[s]| do
12                 (r, b) ← group[s][i]
13                 sig ← (i = |group[s]| − 1) ∨ (i mod batch = 0)
14                 fen ← (i = |group[s]| − 1)
15                 aps.write_seq(r, b.local_addr, sig, fen, b.size)
```

5.4.4 Batch optimization

The cache policy described in § 5.4.1 reduces remote reads by caching object data locally, but cache write-back at commit still issues one RDMA write per dirty cache block. When transactions modify many objects, this creates a burst of individual write operations, each consuming a work queue entry and requiring completion notification. The **batch-cache policy** (Listing 25) extends caching with segmented batching to amortize this overhead.

The `write_back` method in batch_cache_t first groups dirty cache blocks by RDMA segment identifier—the memory node or queue-pair where each block resides. Within each segment, the policy posts a sequence of writes using `WriteSeq`, deferring signaling until the final write in each batch (or every n-th write, where n is tunable via the *batch* parameter). The *sig* flag requests completion notification, while the *fen* flag enforces ordering through a fence operation. This transforms a commit touching 50 dirty blocks from 50 individual writes with 50 completion

events into a handful of batched write sequences, one per segment, with only a few completion events. The optimization reduces both latency (by eliminating wait-for-completion cycles between writes) and NIC queue pressure (by minimizing doorbell operations). The batch-cache policy preserves the same logical semantics as the basic cache policy but achieves higher throughput under write-intensive workloads.

Beyond cache write-back, APS enables batching for ownership record (orec) operations. STM algorithms rely on three fundamental orec operations: validation (checking that orecs have not advanced beyond the transaction's start time), acquisition (locking orecs via CAS), and release (updating orecs to a commit timestamp). In the baseline approach, each operation issues a separate one-sided RDMA verb for every orec. When transactions touch many objects—common in workloads with large read or write sets—this results in hundreds or thousands of individual network operations, each incurring latency and consuming NIC resources.

APS provides an orec batching policy that mirrors the cache batching strategy. When the STM needs to validate or acquire multiple orecs, the batching policy first groups them by memory segment identifier—the RDMA region where each orec resides. Within each segment, orecs are then processed using scatter-gather operations (ReadSeq/WriteSeq provided by Remus) rather than individual verbs. The batching layer manages signaling and fencing: operations within a batch are posted without signaling, and only the final operation in each batch (or every n-th operation) requests completion notification. This transforms a validation of 100 orecs from 100 individual reads into a handful of batched reads, one per segment, dramatically reducing both latency and queue pressure.

Like cache and clock policies, batching is a configurable policy parameter. APS

exposes both a default policy that issues individual operations and a batching policy that groups operations by segment. STM implementations template over these policies, allowing the same transactional logic to run with or without batching depending on workload characteristics. Together with cache, placement, and clock policies, batching completes the set of APS optimizations that enable efficient RDMA-based STM without altering high-level transactional semantics.

5.4.5 Data Structure-Level Optimizations

While APS policies operate transparently beneath the STM interface, data structure programmers can further reduce RDMA overhead through algorithm-aware optimizations. Two techniques prove particularly effective: read-set trimming and cache-aware layout. Neither is an APS policy—both require explicit design decisions at the data structure level—but both significantly impact RDMA performance.

Read-Set Trimming

Pointer-based traversal accumulates large read sets. A skip list search may traverse 20+ nodes, recording each orec in the transaction's read set. At commit time, STM validates every orec, requiring one RDMA read per orec. The STM layer exposes `trim(n)` to prune the read set to its last n entries when semantic invariants permit. The key insight: once an operation's outcome is determined, many traversal reads become redundant. If any concurrent modification affecting the outcome must write at least one orec in the retained set S plus the write set W, then validating only $S \cup W$ suffices. This requires the **cut-set property**: $S \cup W$ forms a cut-set if every execution changing the outcome writes at least one orec in $S \cup W$.

Consider skip list insert-fail where `succ.key = search_key`. Any trans-

action changing this outcome must modify `succ`, writing its orec. Therefore $S = \{\texttt{succ.orec}\}$ and $W = \emptyset$ form a cut-set; `trim(1)` retains only the decisive successor. For insert-success, modified nodes enter W, so `trim(0)` is safe. Mechanically, `trim(n)` discards the prefix of STM's read-set log. At commit, STM validates only the trimmed set. Safety is not automatically verified—programmers must prove the cut-set property. Incorrect trimming violates opacity.

Cache-Aware Layout

Data structure layout choices affect cache-block efficiency. The baseline skip list stores each node's key separately from its forward-pointer array. A search traverses the tower, reading forward pointers at each level and eventually reading the successor's key. With APS caching enabled, reading a forward pointer fetches the node's cache block, but reading the successor's key requires a separate fetch because the key resides in a different object.

A cache-aware layout co-locates the successor key with the forward pointer in the same cache block. When search reads a forward pointer, it implicitly fetches the successor's key in the same RDMA operation. The traversal logic remains unchanged—the algorithm follows the same pointer-chasing pattern—but each cache-block load now delivers both the pointer and the key. This eliminates one cache miss per level. The same principle applies broadly: co-locating frequently-accessed fields, using fixed-size arrays instead of separate pointers, and aligning object boundaries with cache-block granularity all reduce remote fetches without changing algorithmic semantics.

5.5 Evaluation

The evaluation explores how APS policies affect RDMA-based STM performance across varying workloads, data structures, and deployment configurations. The goal is modest and exploratory: to identify which optimization levers matter most, understand when they help, and validate that APS enables these optimizations without altering data structure logic. The evaluation does not claim to be exhaustive or compare against other systems; this is an early-stage architectural study demonstrating that the separation of concerns works and that certain policies consistently improve performance.

5.5.1 Experimental Setup

Platform. All experiments run on CloudLab [161] using r320 nodes connected via 56 Gbps Mellanox FDR InfiniBand. Each node has an Intel Xeon E5-2450 (8 cores, 2.1 GHz) and 16 GB RAM. We use Remus for one-sided RDMA operations (READ/WRITE/CAS/FAA) with preregistered memory and stable remote addresses.

Node roles and configurations. The evaluation considers two deployment patterns:

- **Non-overlapping:** Dedicated memory nodes (passive, expose RDMA regions) and compute nodes (active, issue verbs). Example: 1 memory node + 8 compute nodes. All data resides remotely; every access incurs an RDMA round trip.
- **Overlapping:** Each node serves dual roles as both memory and compute. Nodes can optimize local writes via NIC-bypass (`memcpy`+`clflush`/`sfence`), achieving 2–3.5M ops/sec compared to ∼250–300K ops/sec for remote writes [162, 163].

This configuration yields 25–28% higher throughput in write-heavy workloads due to local-write optimization.

Configuration labels follow the pattern `XXYY_MN##_CN##`, where:

- `XX`/`YY` = insert%/remove% (e.g., `5050` = 50% insert, 50% remove, 0% lookup)
- `MN##` = memory node ID range (e.g., `MN00` = node 0 only; `MN07` = nodes 0–7)
- `CN##` = compute node ID range (e.g., `CN815` = nodes 8–15; `CN07` = nodes 0–7, overlapping)

Workloads. We use standard integer-set benchmarks [38] with three operation mixes:

- **0/0 (read-only):** 0% insert, 0% remove, 100% lookup
- **10/10 (read-heavy):** 10% insert, 10% remove, 80% lookup
- **50/50 (write-heavy):** 50% insert, 50% remove, 0% lookup

Key range is 64K, prefilled to 50% occupancy with disjoint ranges per thread. Each trial runs for 10 seconds. Results are averaged over multiple runs to ensure consistency.

Data structures. We evaluate four pointer-based structures: doubly-linked list, skip list, red-black tree, and hash table. The hash table used in clock experiments is a resizable implementation adapted from prior exotm/STMCAS [19] work (see CaruMap in Figure 5.16).

Concurrency. Compute threads use Boost.Fiber for lightweight cooperative scheduling, multiplexing multiple fibers per OS thread to hide RDMA latency. Context-switch overhead is sub-microsecond (63–165 ns), far below kernel thread

switching. Optimal thread/fiber counts vary by workload and scale; we report the best-performing configuration for each setting.

Metrics. We measure:

- **Throughput:** Total operations per second (aggregated via RDMA atomics)
- **RDMA operation count:** READ/WRITE/CAS/FAA per data structure operation

5.5.2 Experimental Methodology

Each experiment follows a synchronized execution lifecycle:

1. **Initialization:** Leader thread (first compute node, first fiber) allocates shared synchronization structures (global clock, orec tables if needed) and publishes their addresses in RDMA memory. Follower threads read these pointers via one-sided operations.
2. **Data structure initialization:** Leader thread allocates and initializes the data structure root (e.g., list sentinels, tree root); follower threads retrieve root pointers through barriers.
3. **Prefill:** All threads cooperatively insert elements to 50% occupancy using disjoint key ranges (barrier synchronization ensures completion).
4. **Measurement:** All threads execute the workload for 10 seconds, performing operations transactionally according to the configured insert/remove/lookup ratios.
5. **Metrics collection:** Threads aggregate per-thread metrics (operation counts, RDMA statistics) to a shared location using RDMA `FetchAndAdd`. Leader thread reads final metrics and outputs results.

Barriers between phases ensure consistent start/stop times. All synchronization uses APS mechanisms (RDMA atomics, not MPI or external coordination).

5.5.3 Baseline Configuration

Unless otherwise stated, experiments use:

- **STM:** Lazy (commit-time locking, write-back redo logs)
- **Clock:** GV1 (global version via `FetchAndAdd`)
- **Cache:** `nocache_t` (direct RDMA per field)
- **OREC:** Per-object (embedded in object header)
- **Batching:** Disabled (individual RDMA operations)

Each subsection varies one dimension while holding others constant.

5.5.4 Scalability Analysis

We evaluate throughput scaling from 1 to 8 compute nodes using lazy STM with `nocache_t` (direct RDMA, no staging) to isolate concurrency scaling from cache policy effects. This baseline demonstrates raw scaling behavior; cache impact is analyzed in § 5.5.5.

Using the Boost.Fiber configuration described in § 5.5.1, we test configurations up to 8 threads/node with 256 fibers/thread. Context-switch overhead remains sub-microsecond (Figure 5.4).

Results (Figures 5.1, 5.2, and 5.3): All four data structures (doubly-linked list, hash table, red-black tree, skip list) scale across three workloads (0/0 read-only, 10/10 read-heavy, 50/50 write-heavy). Optimal thread/fiber counts vary by scale and workload without consistent patterns—configuration depends on contention,

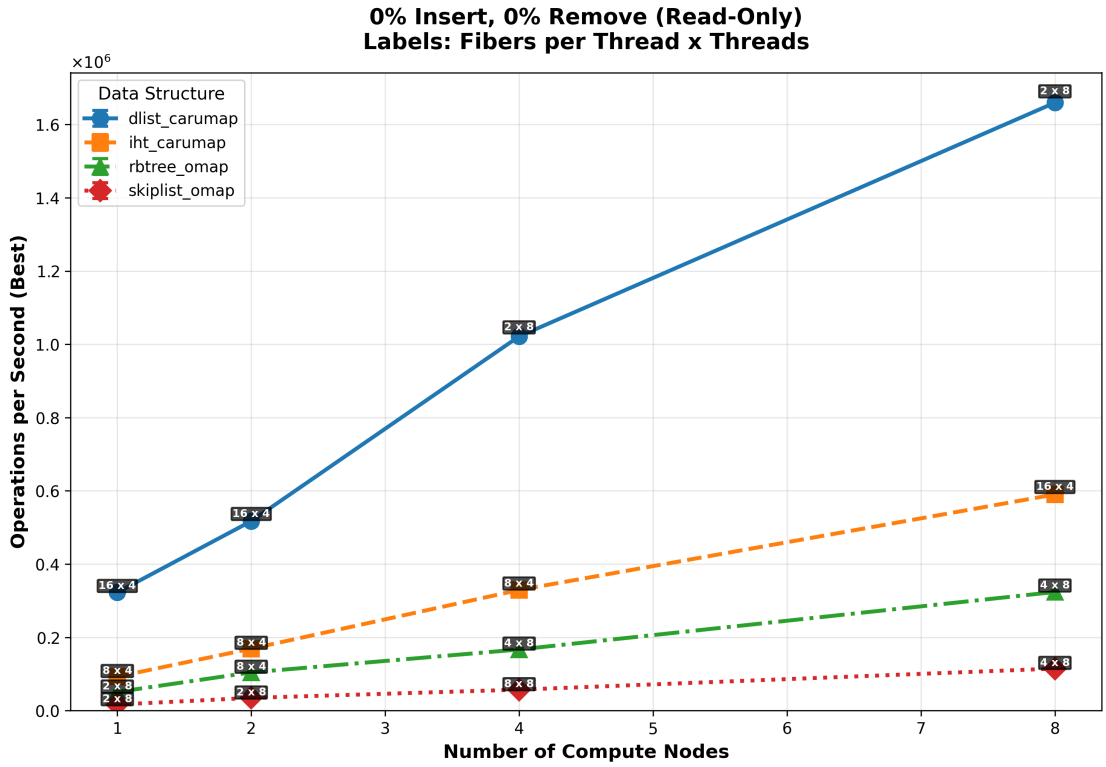


Figure 5.1: Throughput scaling across 1–8 compute nodes for read-only workload (0% Insert, 0% Remove). Labels indicate optimal fibers-per-thread × threads at each scale.

latency, and DS semantics. This baseline characterizes scaling for workload-specific tuning.

5.5.5 Cache Policy Performance

Cache policies exhibit the most dramatic performance impact among all policy dimensions. We compare `nocache_t` (direct RDMA per field) and `cache_t` (local staging with write-back) across three workloads (0/0 read-only, 10/10 read-heavy, 50/50 write-heavy) using red-black tree with varying node sizes (1–32 elements per node).

Setup. Non-overlapping configuration: 1 memory node + 8 compute nodes,

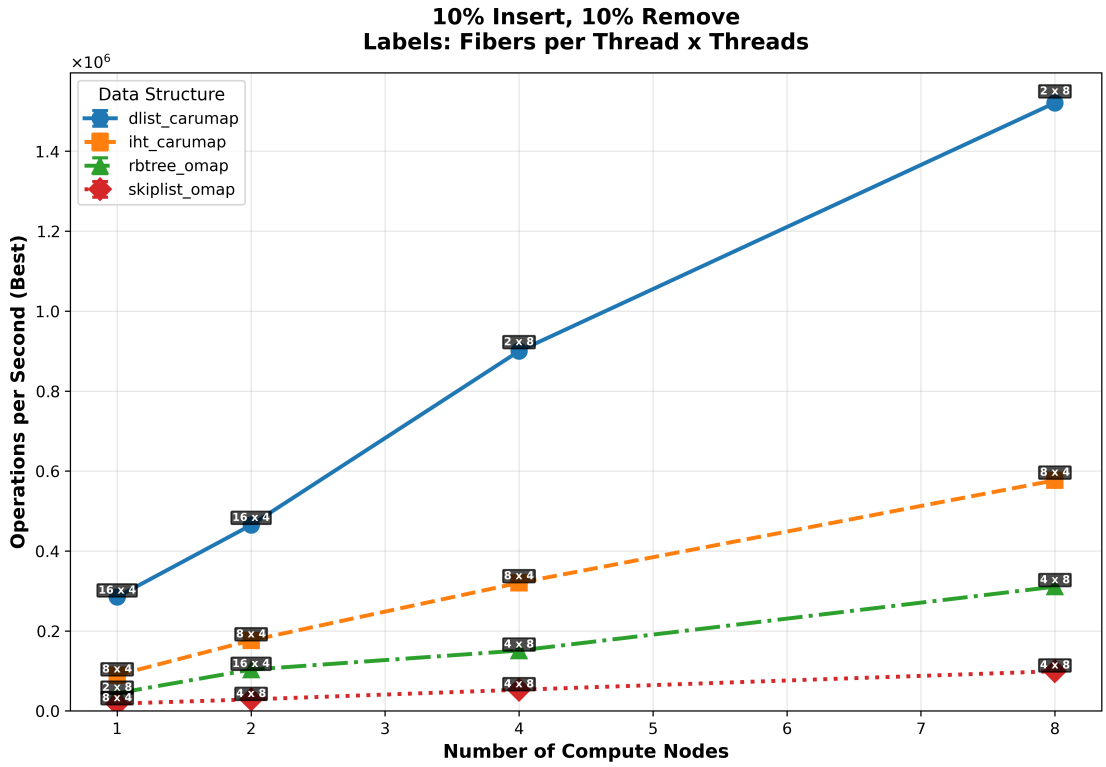


Figure 5.2: Throughput scaling across 1–8 compute nodes for read-heavy workload (10% Insert, 10% Remove). Labels indicate optimal fibers-per-thread × threads at each scale.

8 threads per compute node. Red-black tree, 64K key range, 50% prefill, nodes with 1–32 elements (8 bytes each). Each operation accesses *all* elements in visited nodes. `cache_t` fetches entire objects once; `nocache_t` issues separate RDMA per field. At 32 elements per node, a single visit requires 32 field accesses.

RDMA cost model. Figure 5.5 shows RDMA perftest for non-overlapping configuration: operations/sec remains flat (∼250–300K ops/sec) for small transfers (2B–128B), confirming latency-bound behavior where round-trip time dominates payload size. Caching converts many small operations into few large ones (e.g., 32 field accesses = 32 round-trips vs. 1 round-trip for a 256B object fetch).

Results. `cache_t` dramatically outperforms `nocache_t`, delivering nearly 2× throughput improvement (Figures 5.6, 5.7, and 5.8).

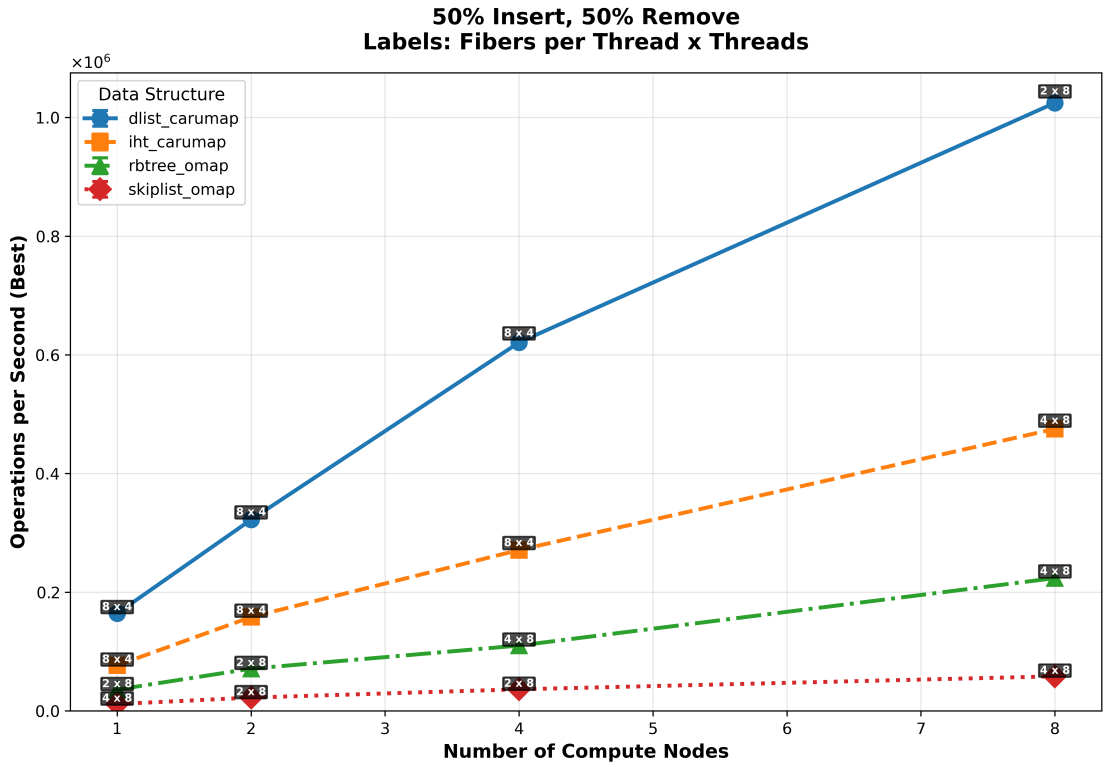


Figure 5.3: Throughput scaling across 1–8 compute nodes for write-heavy workload (50% Insert, 50% Remove). Labels indicate optimal fibers-per-thread × threads at each scale.

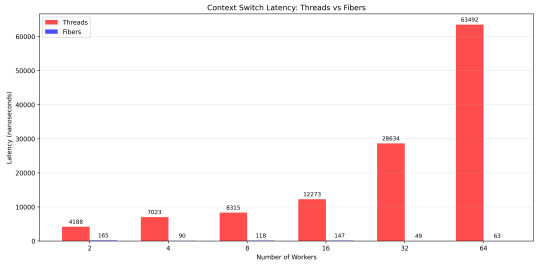


Figure 5.4: Context switch latency: OS threads vs. Boost Fiber cooperative fibers. Fibers maintain sub-microsecond overhead (63–165ns) while thread switching incurs significantly higher cost. Error bars show 95% confidence intervals.

Throughput improvement. Read-only: 230K vs. 120K ops/sec (element size 1); read-heavy: 215K vs. 120K; write-heavy: 155K vs. 80K ops/sec (80–95% improvement).

RDMA reduction mechanism. For read-only workloads, `cache_t` reduces

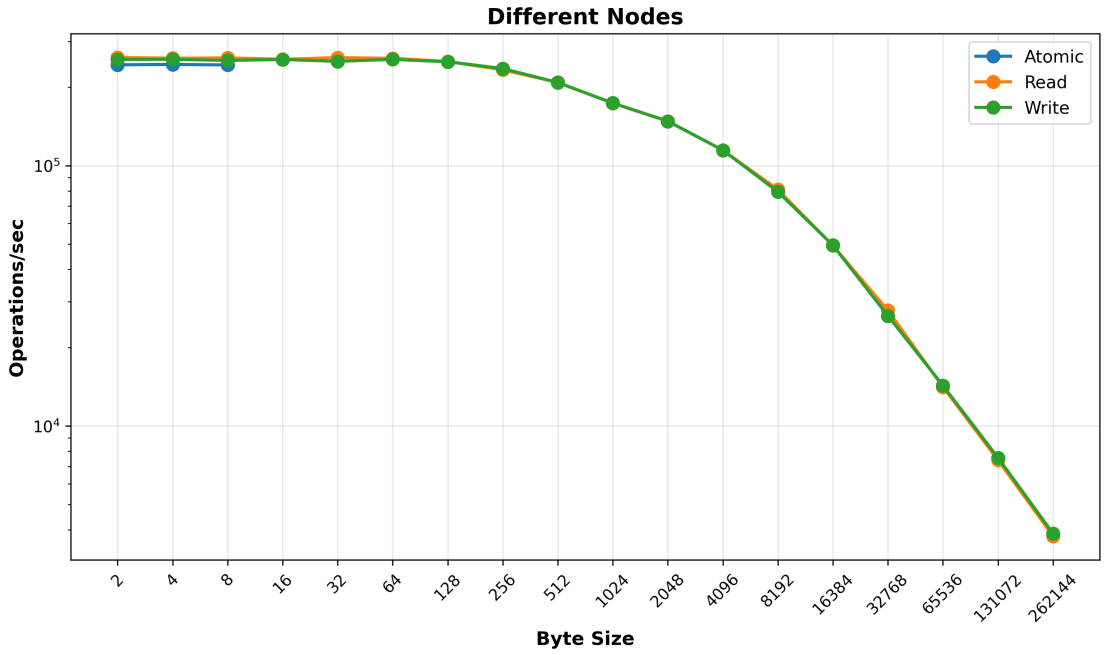


Figure 5.5: RDMA perftest (non-overlapping): latency-bound behavior maintains flat throughput (∼250–300K ops/sec) across small payloads, motivating cache designs that reduce round-trips.

operations from 75–79 to 38 per data structure operation (50% reduction) via: (a) object-level fetching (single RDMA read per object vs. per-field reads), and (b) cache-hit validation optimization—skipping redundant orec validations on already-cached objects (§ 5.4.1).

Latency-bound behavior. Throughput declines as elements per node increase (1→32), but RDMA operations remain constant (`cache_t`: 38–39; `nocache_t`: 75–79), confirming that round-trip count dominates cost, not payload size.

Write-heavy amplification. Cache reduces write-heavy RDMA operations from 103–110 to 49–52 via write-back buffering combined with cache-hit validation, yielding ∼95% throughput gain.

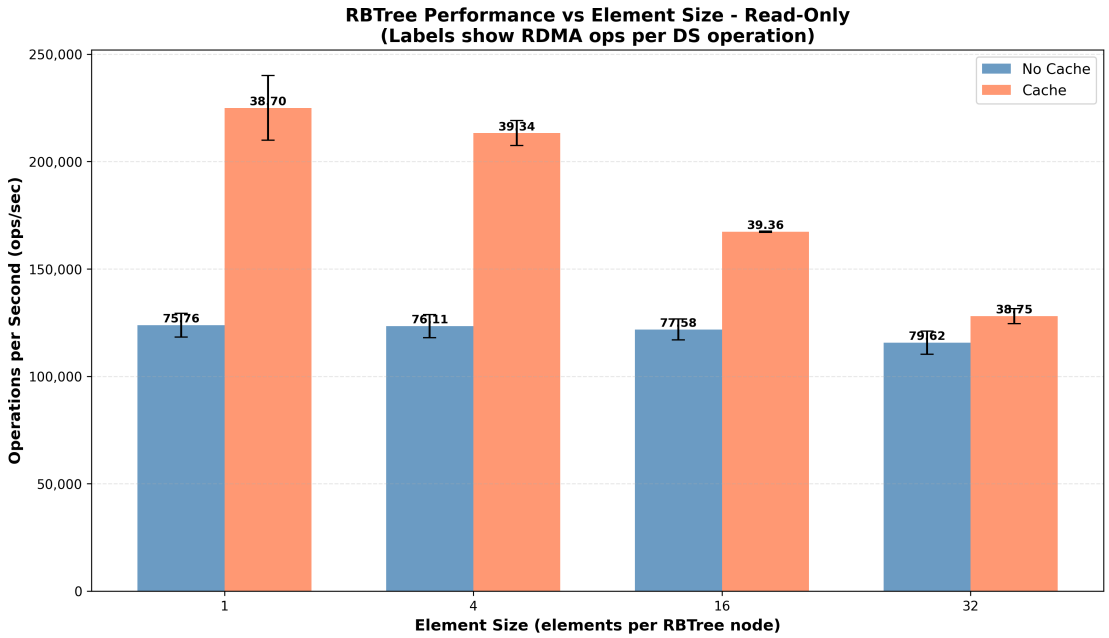


Figure 5.6: Cache policy performance for read-only workload (0% Insert, 0% Remove) on red-black tree with 1 key and 1–32 values per node. Labels show RDMA operations per data structure operation. `cache_t` *achieves nearly 2× throughput via object-level fetching and cache-hit validation optimization.*

Skip list cache optimization

Skip lists expose two orthogonal caching dimensions that interact with the fixed-size object requirement (§ 5.3). We evaluate four configurations (1 MN + 8 CNs, 64K keys, lazy STM) to understand the interplay.

Tower layout pitfall. Skip list nodes contain a forward-pointer tower that varies in height. A naive implementation uses `tower[]` (C flexible array member), allocating tower storage separately via `LOG_NEW` at node creation. However, APS's object-level cache operates on fixed-size types: when caching fetches a node object, it retrieves only the object boundary—the node header with metadata—but *not* the separately-allocated `tower[]` data, because flexible arrays reside outside the object. Each tower access still requires separate RDMA operations, defeating cache

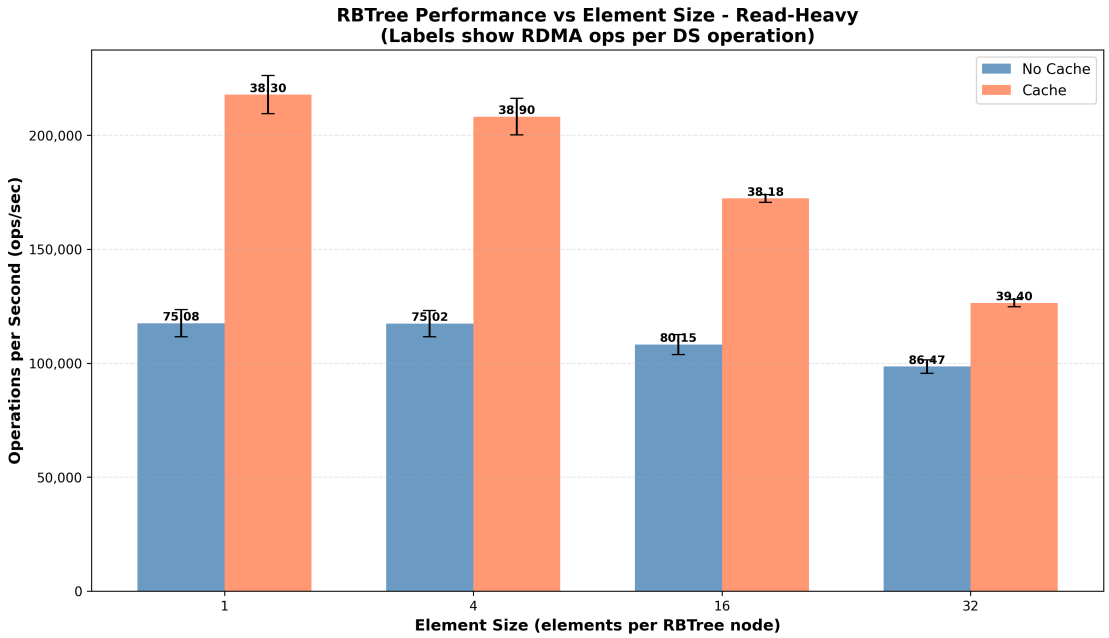


Figure 5.7: Cache policy performance for read-heavy workload (10% Insert, 10% Remove) on red-black tree with 1 key and 1–32 values per node. Labels show RDMA operations per data structure operation. `cache_t` *achieves nearly 2× throughput via object-level fetching and cache-hit validation optimization.*

benefits.

The fix: `tower[16]`, a fixed-size array embedded directly in the node struct. Now the entire tower resides within the object boundary. When cache fetches the node, it gets both header *and* tower in one RDMA read. Subsequent tower accesses hit the cache without additional round trips. This is the pitfall mentioned in § 5.3: variable-length fields break object-level caching.

Key co-location optimization. Independently, skip list traversal normally reads a forward pointer, follows it, then reads the successor's key to decide whether to continue horizontally or drop a level. This requires two RDMA reads per step. By co-locating each successor's key alongside its forward pointer in each tower level (increasing level size from one pointer to pointer+key), traversal can read both in one RDMA operation, enabling early termination without fetching the successor

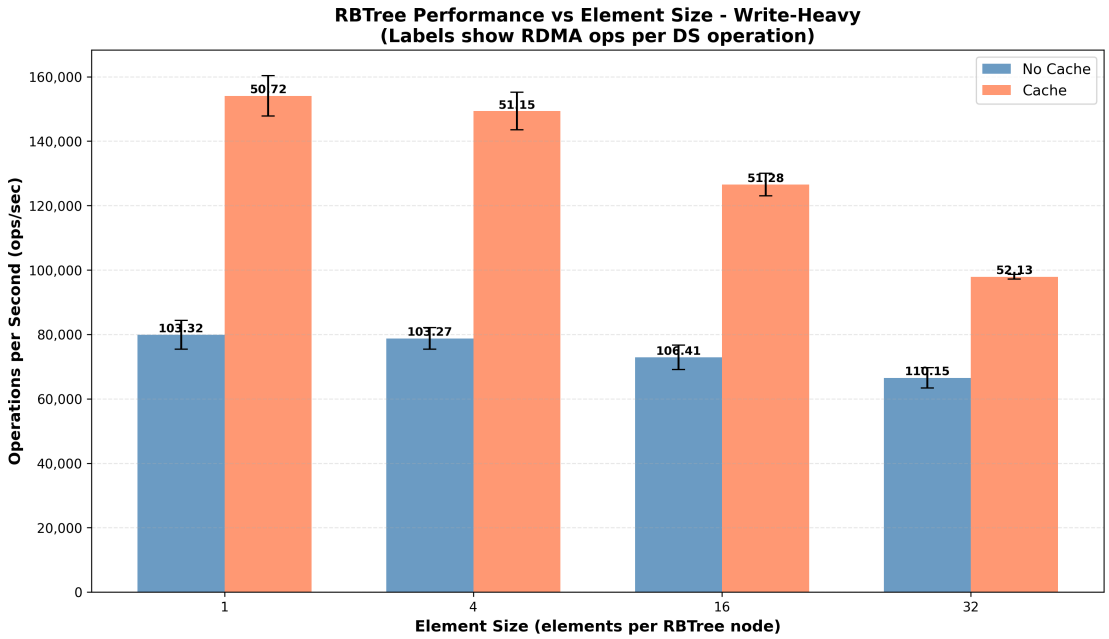


Figure 5.8: Cache policy performance for write-heavy workload (50% Insert, 50% Remove) on red-black tree with 1 key and 1–32 values per node. Labels show RDMA operations per data structure operation. ***`cache_t`*** *achieves nearly 2× throughput via object-level fetching and cache-hit validation optimization.*

node itself.

Four configurations tested:

- `tower[]` + no-key-cache ("No Cache"): Cache fetches only headers; tower and keys require separate RDMA per access
- `tower[16]` + no-key-cache: Cache fetches entire nodes including towers; keys still separate
- `tower[]` + key-cache: Keys co-located in levels (cached with pointers); tower data still separate
- `tower[16]` + key-cache ("Cache-aware"): Cache fetches entire nodes; keys co-located for early termination

Results (Figure 5.9): `tower[16]`+key-cache achieves 38.2 RDMA ops/DS-op

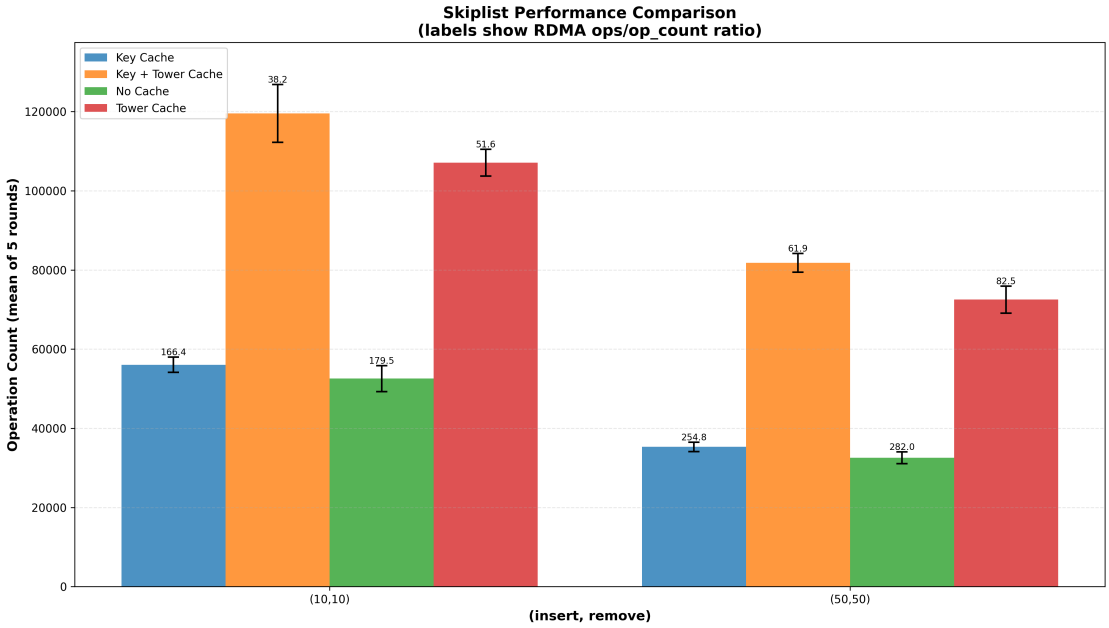


Figure 5.9: Skip list cache-aware optimization: four scenarios (cached/not-cached tower × level-key-cached/not-cached). Labels show RDMA operations per DS operation; bar height is throughput. Cache-aware+enabled achieves lowest operation count via early termination.

(10/10), 61.9 (50/50)—79% reduction vs `tower[]`+no-key-cache (179.5, 282.0), 25% reduction vs `tower[16]`+no-key-cache (51.6, 82.5). Fixed-size layout is essential for cache effectiveness; key co-location provides additional algorithmic gains through early termination.

5.5.6 Batch Operation Effectiveness

The framework batches **orec operations** (validation/acquisition grouped by segment) and **cache write-back** (dirty blocks flushed in segment-aware batches) to reduce RDMA doorbell and fence overhead. We evaluate when batching helps by examining workload characteristics and memory distribution effects.

Local-write optimization. Figure 5.10 shows RDMA microbenchmarks for overlapping configuration (memory local to compute node). The RDMA substrate optimizes local writes to bypass the NIC via `memcpy` + `_mm_clflush`/`_mm_sfence` [162, 163], achieving 2–3.5M ops/sec—far above remote writes. This explains why overlapping experiments reduce batching's relative value.

Policies. We compare four combinations: (1) no batch, (2) orec batch only (`batch_orecs_op_t`), (3) cache batch only (`batch_cache_t`), (4) both batches.

Key insight: Batching effectiveness depends on memory concentration and write intensity. Figures 5.11, 5.12, 5.13, 5.14, and 5.15 show five configurations. Charts report total operations over 10 seconds (proportional to throughput).

Write-heavy vs. read-only (1 MN, non-overlapping). Comparing Figures 5.11 (write-heavy) and 5.12 (read-only), both with 1 memory node + 8 compute nodes: Write-heavy achieves ∼5–10% improvement with batching; orec batching dominates the gain. Read-only shows minimal impact—no cache write-backs or batchable orec validation.

Concentrated vs. distributed memory (write-heavy, non-overlapping). Comparing 1 MN (Figure 5.11) to 8 MNs (Figure 5.13): Single memory node enables large batches (5–10% gain). Distributed memory (8 MNs) fragments batches, yielding negligible benefits (within ±2–3%).

Non-overlapping vs. overlapping (local-write optimization). Overlapping configurations (Figures 5.14, 5.15) achieve 25–28% higher throughput (1 MN: ∼36–37K vs. ∼29K ops/sec) via local-write optimization (NIC bypass). Batching

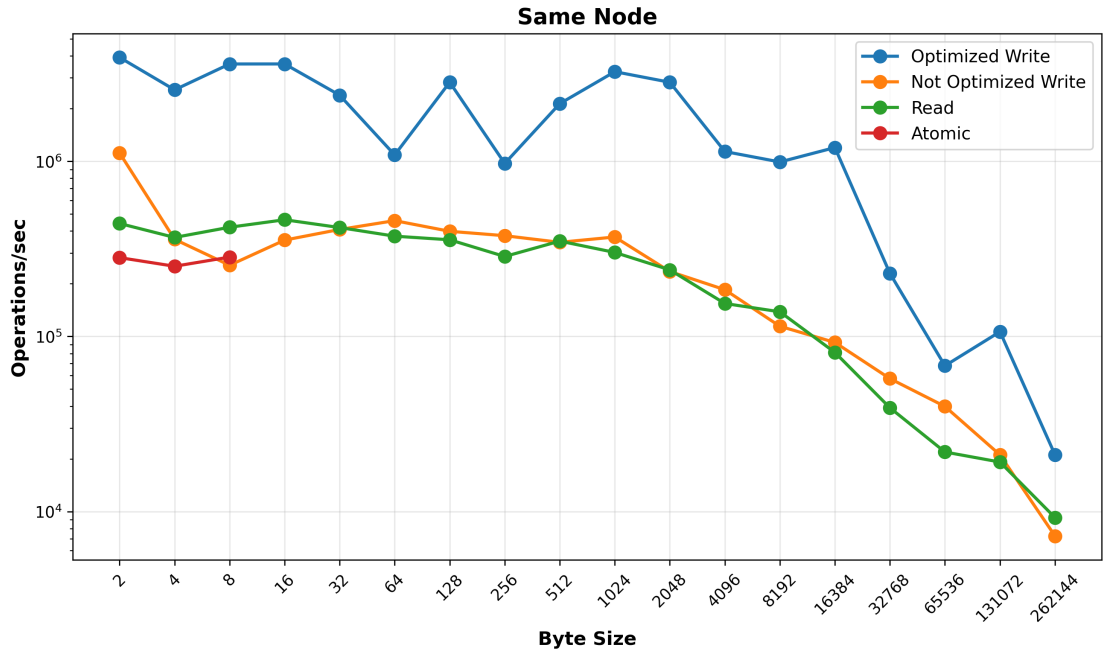


Figure 5.10: RDMA perftest (overlapping): optimized writes bypass NIC, reaching ∼2–3.5M ops/sec, demonstrating local-write optimization.

benefit shrinks to ≤4% (vs. 5–10% non-overlapping) because local writes eliminate most doorbell/fence overhead.

Summary. Write-heavy + concentrated memory (1 MN, non-overlapping): batching improves throughput 5–10% (orec batching dominates). Read-only or distributed memory (8 MNs): negligible benefits (within ±2–3%). Overlapping: local-write optimization boosts throughput; batching adds ≤4%.

5.5.7 Clock Policy Performance

We evaluate four clock policies (GV1, Funnel GV1, GV5, Funnel GV5) using a resizable hash table under write-heavy workloads (50% insert/remove, 64K key range, 2–16 threads on 8 nodes). Policies differ in commit strategy (GV1: FAA; GV5: read+1) and local caching (funnel variants cache timestamps per-node; see

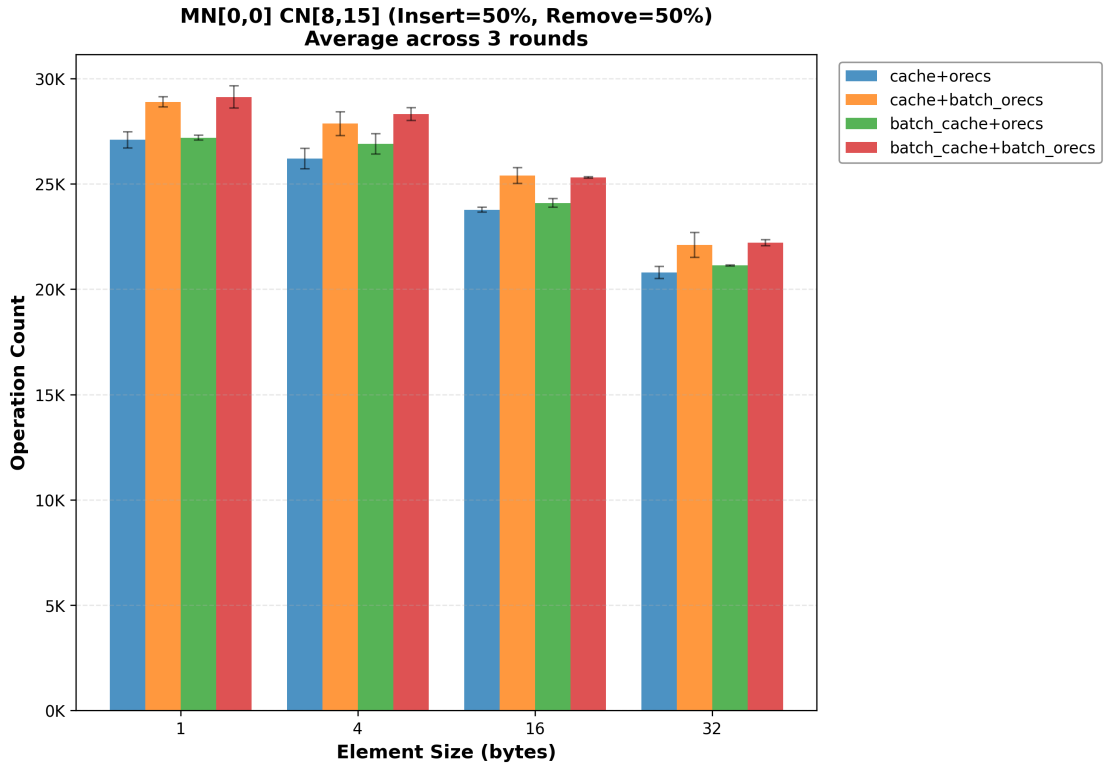


Figure 5.11: Batch operation effectiveness: Write-heavy (50% Insert, 50% Remove) with 1 memory node, non-overlapping (5050_MN00_CN815). Four policies: no batch, orec batch, cache batch, both batches. Batching improves throughput 5–10% with orec batching dominating.

§ 5.4.3). This high-contention scenario maximizes clock pressure: every transaction commits and frequent conflicts amplify clock accesses. Results are normalized to GV1; bar labels show RDMA operations per data structure operation (Figure 5.16).

Results. All four policies achieve similar throughput (within 20%) despite maximum clock pressure, with RDMA ratios consistently around 22–24 operations per data structure operation. Data structure operations (node reads, value updates, orec management) dominate clock costs, as each transaction performs at most 2 remote clock accesses.

Funnel GV5 achieves the best performance with the lowest RDMA operation count. Funnel variants provide the largest benefit for GV5 by caching timestamps

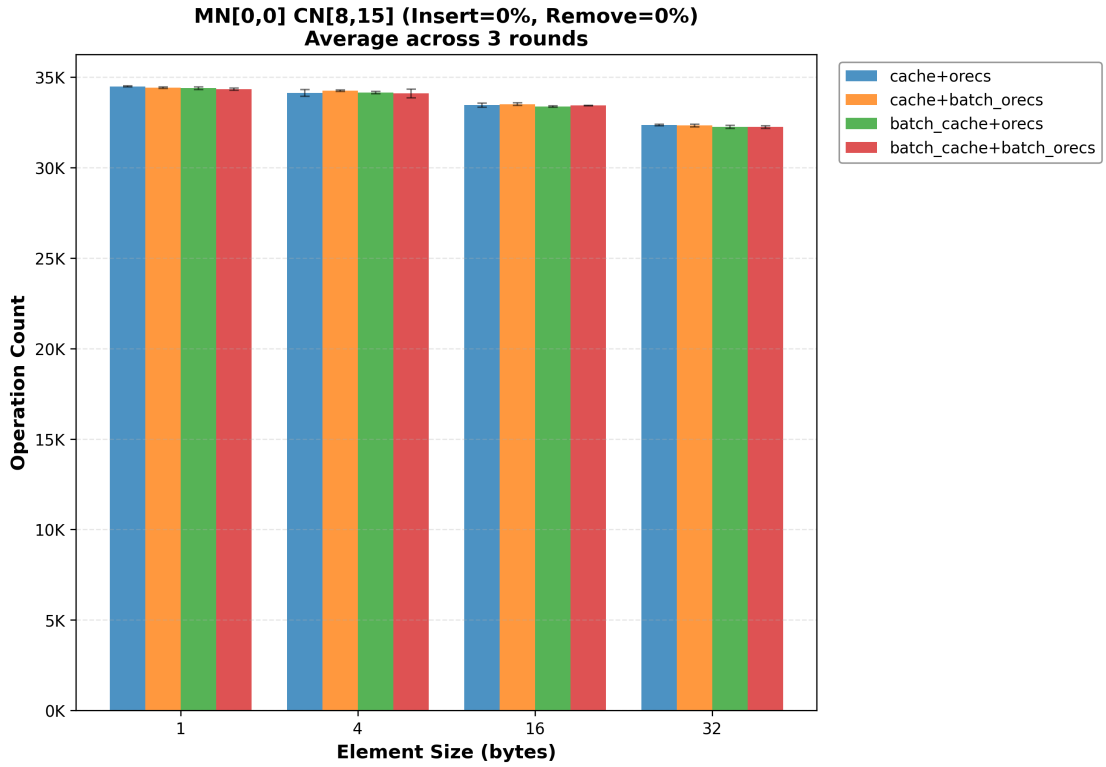


Figure 5.12: Batch operation effectiveness: Read-only (0% Insert, 0% Remove) with 1 memory node, non-overlapping (00_MN00_CN815). Four policies: no batch, orec batch, cache batch, both batches. Batching shows minimal impact due to lack of write-backs.

locally and avoiding redundant remote reads, demonstrating that local caching effectively amortizes clock access overhead in write-intensive workloads.

5.5.8 Readset Trimming Optimization

The `trim(n)` API allows data structure programmers to reduce readset to the last `n` elements when semantic knowledge permits (§ 5.4.5), reducing commit-time validation overhead.

Methodology. STM instrumentation collects: pre/post-trim readset size, validation RDMA operations, abort rate, and commit latency. Configuration: 1 MN + 8 CNs, 64K key range, 50% prefill, three workloads (0/0, 10/10, 50/50). We

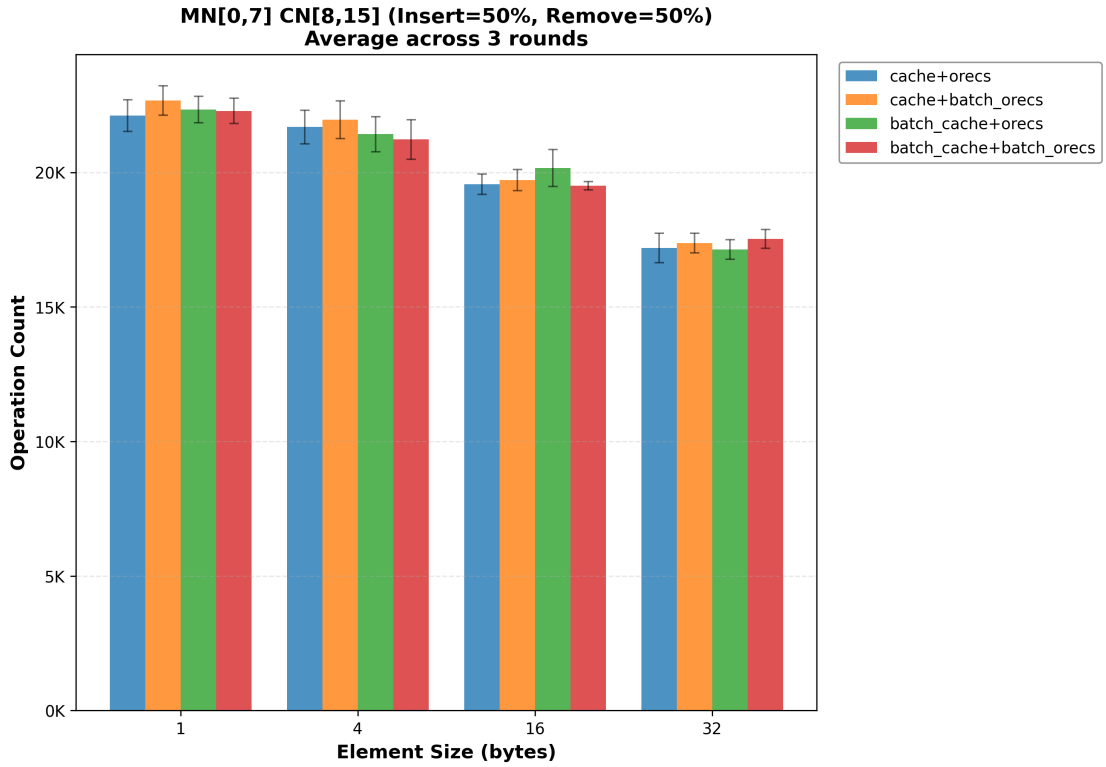


Figure 5.13: Batch operation effectiveness: Write-heavy (50% Insert, 50% Remove) with 8 memory nodes, non-overlapping (5050_MN07_CN815). Four policies: no batch, orec batch, cache batch, both batches. Distributed memory fragments batches, yielding negligible benefits.

compare trim-enabled vs. trim-disabled.

Trimming strategies. Skip list: Insert-fail `trim(1)` (matched node); insert-succeed `trim(0)` (all prev/succ acquired); remove-fail `trim(1)` (succ's orec); remove-succeed `trim(0)`. **Red-black tree**: Insert-fail `trim(1)` (matched node); insert-succeed `trim(0)` (modified nodes acquired); remove-fail `trim(1)` (parent); remove-succeed `trim(0)`.

Results (Figure 5.17, write-heavy 50/50). **RDMA reduction**: `rbtree_omap` reduces from 50.35 to 42.70 ops/DS-op (∼15% reduction); `skiplist_omap` from 80.33 to 58.57 (∼27%). **Throughput**: `rbtree_omap` improves from 161,102 to 187,102 ops/s (∼16%); `skiplist_omap` from 75,120 to 97,781 ops/s (∼30%). **Mech-**

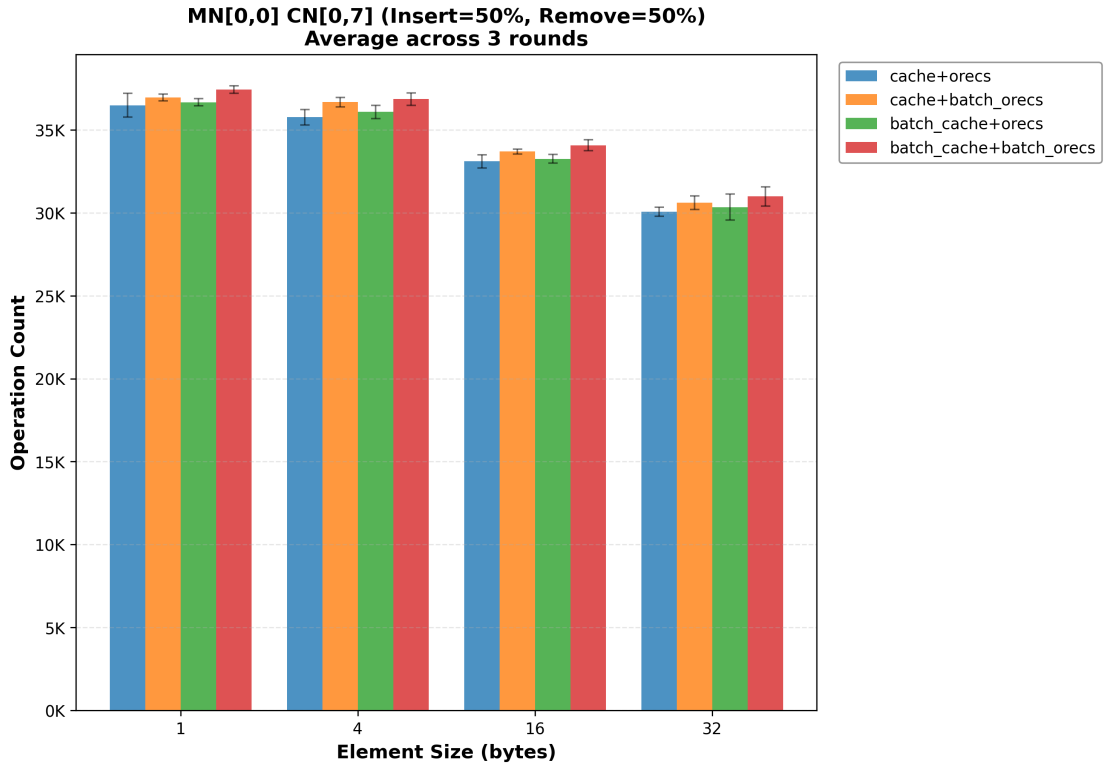


Figure 5.14: Batch operation effectiveness: Write-heavy (50% Insert, 50% Remove) with 1 memory node, overlapping (5050_MN00_CN07). Four policies: no batch, orec batch, cache batch, both batches. Local-write optimization boosts throughput 25–28%; batching adds ≤4%.

anism: Trimming prunes validation set after outcome determination (§ 5.4.5), cutting commit-time orec validations.

The optimization combines with cache policies for cumulative benefits. Read-heavy and read-only workloads show minimal benefit (few/no lock acquisitions limit trimming opportunities).

5.5.9 Discussion

Across all experiments, **object-level caching delivers the largest performance gains** (nearly 2× throughput), reducing RDMA round trips by 50–80% through

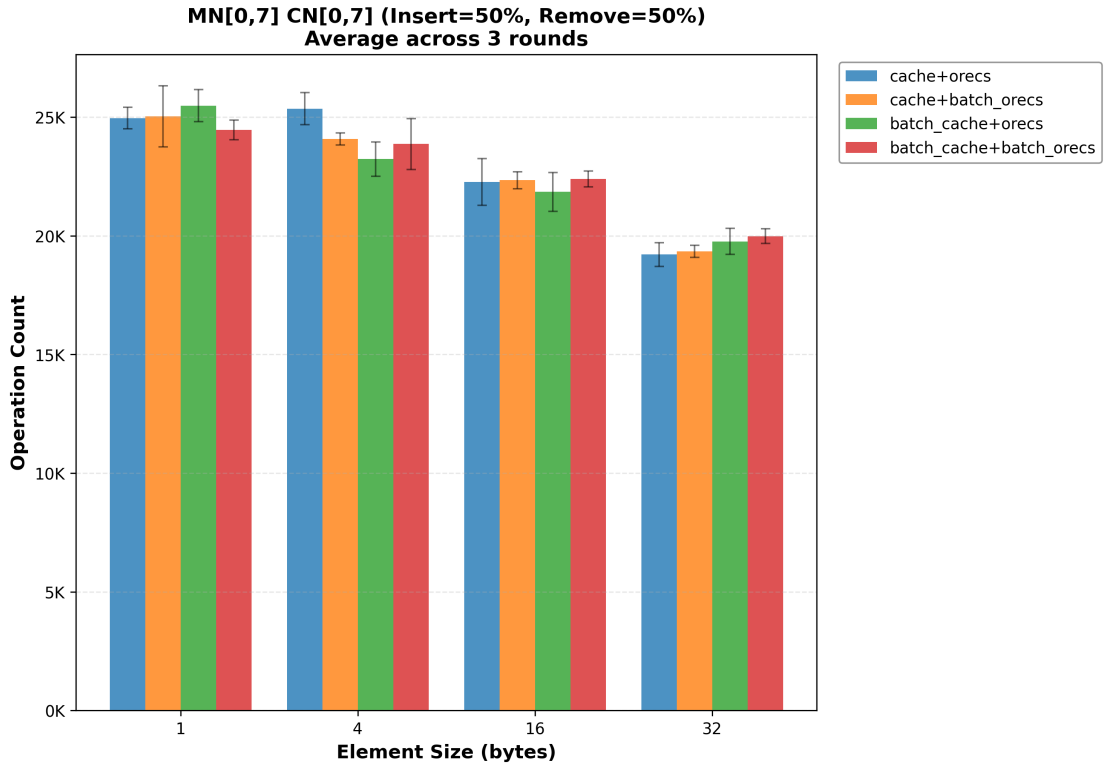


Figure 5.15: Batch operation effectiveness: Write-heavy (50% Insert, 50% Remove) with 8 memory nodes, overlapping (5050_MN07_CN07). Four policies: no batch, orec batch, cache batch, both batches. Local-write optimization with distributed memory shows minimal batching benefit.

whole-object fetching and cache-hit validation skipping. However, caching effectiveness requires careful data structure layout: variable-length fields (e.g., skip list `tower[]`) reside outside object boundaries and defeat caching, while fixed-size layouts (e.g., `tower[16]`) enable entire structure fetching (§ 5.5.5). This demonstrates that APS's cache policy exposes an architectural requirement—programmers must structure data to exploit spatial locality—rather than transparently handling all layouts. Read-set trimming provides orthogonal benefits (16–30% improvement) when data structure semantics permit. Clock and batching optimizations are workload-dependent and second-order: clock variants differ by <20%; batching helps only when memory is concentrated (1 memory node) and workloads are

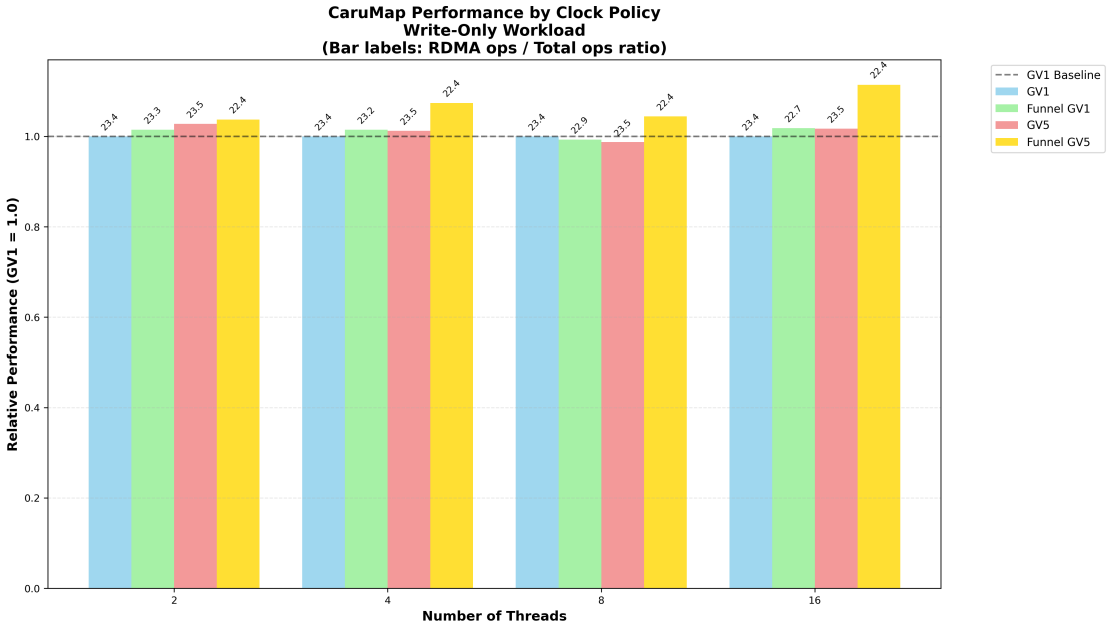


Figure 5.16: Clock policy performance on CaruMap (write-heavy, 50%/50%). Normalized to GV1; labels show RDMA operations per data structure operation.

write-heavy (5–10% gain), with negligible impact when memory is distributed or workloads are read-dominated.

These results demonstrate a key finding: **different policy dimensions have drastically different performance impacts**. Caching delivers nearly 2× gains; trimming provides orthogonal benefits; clock and batching effects are workload-dependent. This heterogeneity motivates APS's systematic exploration approach: rather than prescribing universal solutions, APS enables identifying which levers matter most for specific workloads. Policies are pluggable, data structure logic remains unchanged, and STM correctness is preserved.

5.5.10 Threats to Validity

Our evaluation is deliberately narrow and exploratory. We instantiate one synchronization mechanism (lazy STM), one fabric (RDMA over InfiniBand), and one

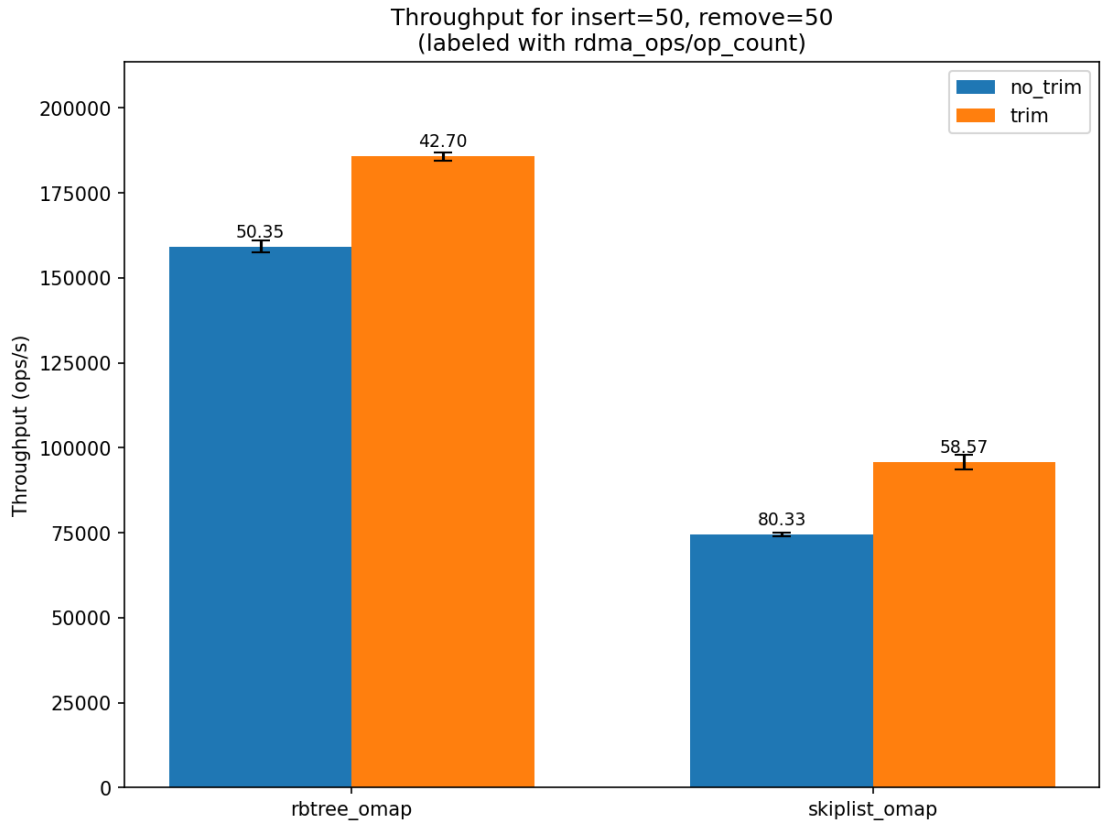


Figure 5.17: Readset trimming (write-heavy, 50/50). Bar height: throughput (ops/s); labels: RDMA ops per DS operation. Trimming reduces operations 15–27% and improves throughput 16–30%.

set of benchmarks (integer sets with uniform key distributions). Broader threats include:

- **Limited workloads:** Skewed distributions, range queries, and application-specific access patterns may yield different results.
- **Single synchronization layer:** Eager STM, lock-free mechanisms, or multi-word CAS may benefit differently from APS policies.
- **Fabric-specific:** CXL, persistent memory, and other fabrics have different latency/bandwidth profiles that may shift policy trade-offs.
- **No external baselines:** We compare policies within our framework but not

against other RDMA STM systems or hand-optimized data structures.

These limitations are acceptable for an architectural study focused on separation of concerns and policy exploration. Broader evaluation and comparison with other systems remain future work.

5.6 Related Work

This work is positioned relative to specialized RDMA transaction systems, high-level abstractions, STM foundations, and emerging memory fabrics.

RDMA transaction systems. FaRM [52] uses one-sided RDMA for distributed transactions with optimistic concurrency control, achieving high throughput through careful protocol design. DrTM [155] and DrTM+H [164] combine RDMA with hardware transactional memory, systematically comparing RDMA primitives for transaction protocols. NAM-DB [165] explores RDMA-aware data structures for main-memory databases. These systems achieve high performance through co-design of synchronization and RDMA mechanics, but their optimizations are tightly coupled—changing workloads or hardware often requires re-engineering core protocols.

This approach differs in focus and scope. The chapter provides a layered architecture that decouples correctness from optimization while enabling systematic exploration through pluggable policies. Where FaRM or DrTM optimize specific design points, APS enables systematic exploration of the policy space (caching, batching, clock strategies, placement) without changing data structure code. The evaluation is deliberately exploratory rather than exhaustive, demonstrating that separation of concerns works and identifying which optimization levers matter most.

Message-based alternatives. eRPC [51] demonstrates that carefully-tuned RPC over commodity networks can match RDMA performance for request-response workloads. FaSST [158] and HERD [156] achieve portability by abstracting RDMA behind high-level APIs, but sacrifice fine-grained control over memory placement and operation grouping. The design assumes one-sided primitives benefit workloads where spatial locality and caching can collapse pointer-chasing into bulk transfers. The evaluation (§ 5.5.5) shows cache-aware layout delivers nearly 2× throughput through object-level fetching combined with cache-hit validation optimization, supporting this assumption.

STM foundations and policy separation. TL2 [42] pioneered time-based STM with global version clocks; the framework adopts its read-validate-commit pattern. SwissTM [166] and TinySTM [61] demonstrate modular STM designs with configurable contention management and validation strategies. STMCAS [19] (ExoTM) exposes explicit validation/acquisition scopes, giving programmers fine-grained control over orec operations.

The framework extends this philosophy to remote memory. Where ExoTM exposes explicit operations, the framework provides *configuration-time control* (pluggable APS policies: cache/clock/batch/orec-placement requiring no data structure code changes) and *programming-time control* (data structure-specific optimizations: cache-aware layout, read-set trimming, anchors). This adapts ExoTM's control exposure to RDMA's optimization landscape while preserving correctness through layered separation. The APS abstraction should generalize beyond STM to k-CAS, LLX-SCX, and other synchronization mechanisms that share the collect-validate-publish pattern.

RDMA design guidelines. Studies by Kalia et al. [50, 158] and Mitchell et al. [167] provide design guidelines for RDMA performance optimization. The evaluation demonstrates a complementary insight: different optimization dimensions have drastically different performance impacts—object-level caching yields nearly 2× throughput (§ 5.5.5), while clock strategies differ by only 20% (§ 5.5.7). These works emphasize *guidelines*; the contribution is a *layered architecture* (APS) that enables systematic exploration to identify which optimization levers matter most for specific workloads, making policies pluggable and composable.

Emerging memory fabrics. CXL studies [168] show that even with hardware coherence, software must engineer locality at object/segment granularity—reinforcing the motivation. The APS abstraction is designed for retargetability: the same layered separation should hold across substrates, with only the APS implementation changing. On CXL, `read`/`write` map to loads/stores, `stage()` becomes prefetch, and anchors lean on coherence; on persistent memory, writes add flush operations. The data structure and STM layers remain unchanged—the intended portability benefit of layered design.

5.7 Conclusion

This chapter presents a layered architecture for remote-memory data structures that separates application/data-structure logic, synchronization semantics, and hardware access. The Access & Placement Substrate (APS) sits between data structures and the synchronization layer, exposing optimization capabilities—object staging, validation optimization, batching, clock management, and placement control—while hiding hardware-specific details. The synchronization layer invokes APS to realize

policies over the hardware fabric. Data structures program against familiar STM interfaces; optimizations are selected via pluggable policies requiring no algorithm changes.

Our prototype instantiates this design with lazy STM over RDMA, implementing APS capabilities for caching, batching, clock strategies, and placement policies. On CloudLab deployments (1–8 nodes, standard integer-set benchmarks, uniform distributions), we explored how these levers affect performance. Object-level caching delivered the largest gains (nearly 2× throughput, 50–80% RDMA reduction) by collapsing pointer-chasing and skipping redundant validations. Read-set trimming provided orthogonal benefits (16–30% improvement) when data structure semantics permitted. Clock and batching optimizations showed workload-dependent effects: clock variants differed by <20%; batching helped (5–10%) only when memory concentrated and workloads were write-heavy.

These results demonstrate that different policy dimensions have drastically different performance impacts—caching nearly 2×, trimming 16–30%, clock <20%, batching 5–10% (concentrated) to ±2–3% (distributed). This heterogeneity underscores the value of APS's systematic exploration approach: rather than prescribing optimal solutions, APS enables programmers to identify which levers matter most for their specific workloads. Policies are pluggable, data structure logic remains unchanged, and STM correctness is preserved.

5.7.1 Design Principles

Three practical lessons emerge from this work:

First, locality is a software responsibility—and layout matters. Hardware fetches cache lines and pages; pointer-rich structures rarely align with those

boundaries. Staging entire objects and validating them once per transaction collapsed dependent round trips. However, object-level caching requires fixed-size layouts: variable-length fields (e.g., flexible arrays) reside outside object boundaries and defeat caching. Programmers must structure data accordingly—embedding skip-list towers as fixed-size arrays, co-locating successor keys with pointers—to exploit spatial locality. One well-placed fetch then becomes several useful reads.

Second, the synchronization layer should leverage APS capabilities and surface semantics-preserving conveniences. After the first orec check on an object, subsequent field reads can use the staged copy; final validation at commit catches intervening writers. Read-set trimming, when data structure semantics permit, reduces commit-time validation overhead without compromising correctness.

Third, group work when it naturally aims at the same place. Batching reduced doorbell traffic when writes concentrated on few destinations; when they scattered, batching neither helped nor hurt. The rule of thumb: separate concerns cleanly through layers, not ad-hoc hooks. Each layer depends on stable semantics from below, enabling independent evolution.

5.7.2 Limitations and Scope

This study is exploratory, not exhaustive. Evaluation covered standard integer-set microbenchmarks (insert/remove/lookup) on four data structures (lists, hash tables, trees, skip lists) using CloudLab r320 nodes with ConnectX-3 NICs. Real workloads—key-value stores with skewed access, array-based B-trees, graph traversals—may show different locality and contention profiles. Most policies were measured independently rather than in full factorial combinations, leaving inter-

action effects unexplored. The platform is older; newer NICs and topologies may shift where optimizations pay off. These boundaries are deliberate: the goal is to make systematic exploration easy, not to prescribe universal recipes.

5.7.3 Future Directions

Several directions could extend this work:

Broader evaluation. The evaluation tested policies independently on standard integer-set benchmarks with uniform distributions. Future work should explore: (a) factorial combinations of policies to understand interaction effects, (b) diverse workloads including skewed access patterns, range queries, and application-specific traces, (c) modern hardware including newer NICs (ConnectX-5/6/7) with improved batching and signaling capabilities, and (d) array-based data structures (B-trees, skip vectors) that amplify staging benefits through better spatial locality.

Other synchronization mechanisms. Horizontally, other synchronization layers (k-CAS, LLX-SCX, versioned lock-free algorithms) share the collect-validate-publish pattern and should map straightforwardly to APS. These mechanisms differ mainly in how updates are published; the staging, batching, and clock management policies should transfer directly. Implementing these would test how well the layered separation generalizes across synchronization paradigms.

Other memory fabrics. Vertically, retargeting APS to emerging fabrics would test portability: on CXL, `read`/`write` become loads/stores and `stage()` becomes prefetch, with anchors leveraging cache coherence; on persistent memory, writes add `clflush`/`clwb` and persist fences, with recovery protocols layered above APS. In

both cases, data structure code should remain unchanged, with only APS requiring a new backend.

Asynchronous operations and prefetching. Issuing prefetch calls and awaiting them later could overlap network latency with useful work. While our measurements suggest removing redundant validations already captures much of the benefit, a careful async design might push further by pipelining RDMA operations. This requires APS to expose ordering guarantees and the synchronization layer to validate prefetched data before use.

Adaptive runtime policies. Most policies are selected at configuration time. A runtime that monitors cache hit rates, contention patterns, and access distributions could dynamically switch policies (e.g., migrating hot orecs, adjusting batch sizes, toggling between clock strategies) without changing data structure code. This would exploit the pluggable policy design while adapting to workload shifts.

Hardware extensions. SmartNICs and DPUs could offload clock maintenance, batched validation, or bulk writes, reducing CPU involvement while preserving the layered architecture. Wider atomic operations (e.g., 128-bit atomics for multi-orec validation) would reduce validation costs by grouping orec checks into fewer operations; these would slot cleanly under APS as new primitives.

The architecture is less a "framework" than a way to keep concerns separated: APS gives the synchronization layer leverage without leaking hardware details; the synchronization layer gives data structures stable semantics they can program against. This separation should help make results reproducible and insights transferable to future memory fabrics.

Chapter 6

Conclusion

This dissertation demonstrates that the false dichotomies long accepted in concurrent programming—programmability versus performance, generality versus efficiency, composition versus specialization—are not fundamental but rather artifacts of inadequate information exposure at layer boundaries. *Architectural decoupling*—separating low-level synchronization primitives from high-level programming abstractions through stable contracts that expose semantic invariants—enables systems to achieve both programmability and performance through incremental optimization. The three projects presented in this work—exoTM/STMCAS, Harmony, and RDMASTM—validate this principle across shared memory, multi-structure composition, and distributed RDMA. By exposing the right semantic information (orec values as proofs, co-located but distinct ownership and versioning metadata, object layout control with validation optimization and temporal locality hints), these systems separate what must be correct (synchronization semantics) from what must be efficient (implementation strategies), demonstrating that programmers need not choose between ease of use and performance, nor between correctness and adaptability. Instead, by identifying the right abstractions and contracts, we can

build systems that are correct by default yet remain open to targeted, incremental optimization.

6.1 Summary of Contributions

The central thesis of this dissertation is that tight coupling between synchronization mechanisms and high-level programming interfaces , while traditionally accepted as necessary for performance, unnecessarily constrains both programmability and optimization opportunities. Each of the three main contributions challenges the conventional wisdom in a different domain, progressively expanding the scope and applicability of architectural decoupling .

6.1.1 exoTM/STMCAS: Establishing the Foundation

The exoTM/STMCAS work (Chapter 3) established the foundational principle by demonstrating that a minimal synchronization substrate—ownership records and a global clock—can simultaneously support multiple usage policies. By isolating the low-level mechanisms that protect orecs and manage version numbers from the usage styles that build upon them, we enabled programmers to choose between full STM semantics (for ease and generality) and STMCAS-style optimized steps (for performance). Critically, these policies can coexist: the same data structure can use STM for complex, uncommon operations while employing STMCAS for performance-critical paths.

The evaluation demonstrated that this flexibility does not come at the cost of performance. STMCAS-based data structures usually outperform strong hand-tuned baselines (lock-based and lock-free CDSs), reaching up to 1.35× their throughput

on specific workloads while preserving STM programmability where needed. This result validated the core hypothesis: architectural decoupling does not prohibit either programmability or high performance—systems can provide both.

6.1.2 Harmony: Extending to Composable Operations

Building on the foundational substrate concepts from exoTM, the Harmony system (Chapter 4) extended architectural decoupling to address composition across multiple data structures. Traditional transactional memory systems couple transaction management, data structure synchronization, and program data access, leading to false conflicts when structural changes are mistaken for semantic conflicts. Harmony introduced a crucial insight: by using separate but co-located metadata for low-level synchronization and high-level transaction management, we can decouple these concerns without sacrificing efficiency.

This separation enabled several innovations. First, semantic and structural versions allow distinguishing between true transaction-level conflicts and benign structural modifications. Second, Harmony's ownership model permits traversals to proceed across nodes being inserted when the operation's preconditions are satisfied, reducing false conflicts without sacrificing safety. Third, C++ coroutines enable program data management to be completely orthogonal to synchronization—each operation yields its result, and the transaction manager triggers cleanup without maintaining explicit undo/redo logs.

Evaluation on microbenchmarks and TPC-C demonstrated that Harmony provides best-in-class performance while being substantially easier to program than prior approaches. Across these workloads, Harmony matches or exceeds Medley, the current state-of-the-art transactional data structure library.

In addition, Harmony contributes three compatible techniques for expressing non-existence in sets and maps, enabling range queries and reducing false conflicts. The system supports complex data structures (skip lists, resizable arrays, deques, hash tables) and complex operations (read-only and mutating range queries).

Importantly, the design achieves formal serializability guarantees with proof sketches showing that concurrent non-commuting operations from different transactions correctly manifest as detectable conflicts.

6.1.3 RDMASTM: Exploring RDMA Through Layered Architecture

The RDMASTM framework (Chapter 5) uses STM as an exploratory testbed to systematically study optimization opportunities for RDMA-based concurrent data structures, where the lack of hardware-managed caching for remote memory and per-operation NIC costs introduce fundamentally different challenges than shared memory. The framework's primary contribution is its layered architecture with an Access & Placement Substrate (APS). APS sits between data structures and the synchronization layer, exposing pluggable policies—object-level caching, validation optimization, batching, clock management, and placement control—while hiding hardware-specific details. The synchronization layer invokes APS to realize policies over the hardware fabric. Data structure algorithm logic remains unchanged; programmers add structural optimizations like anchors for hot metadata or cache-aware layouts for spatial locality.

This exploratory approach enables systematic investigation of how different strategies affect RDMA costs (locality management, per-operation overhead, validation). Data structure code uses the stable field-level API, an STM layer handles

concurrency control and logging, and pluggable southbound services implement RDMA-specific access strategies underneath. Our exploratory evaluation on CloudLab (1–8 nodes, four data structures, three workloads) demonstrates that the framework enables meaningful investigation of RDMA optimization opportunities. However, the evaluation has important limitations: most optimizations were tested independently rather than in factorial combinations; workloads focused on standard integer-set operations with uniform random access patterns. These findings illuminate promising directions while highlighting the need for more comprehensive evaluation. The exploratory results reveal three patterns. First, object-level caching with validation-skip optimization shows substantial impact. Caching reduces RDMA operations by 50–80% and improves throughput by nearly 2× across workloads by collapsing pointer-chasing and skipping redundant orec checks for cached objects. Data structures can co-design with caching: cache-aware skip lists co-locate successor keys with pointers, enabling early termination optimizations that further reduce RDMA costs. Readset trimming provides orthogonal gains (16–30% throughput improvement) when data structure semantics permit safe validation pruning. Second, clock strategies show smaller variance (within <20%) but still provide measurable tuning opportunities, with Funnel GV5 achieving the best performance under our workloads. Third, batching effectiveness depends on memory access distribution: 5–10% improvement for write-heavy workloads with concentrated memory, minimal benefit (within ±2–3%) when memory is distributed. The system scales to 8 nodes; throughput increased by up to about 5× from 1 to 8 nodes (workload- and structure-dependent). Beyond these specific findings, the work derives general principles for RDMA concurrent data structures—explicit locality management through object-level caching, validation optimization via

cache-hit tracking, and operation batching for amortizing per-verb costs—that likely generalize beyond STM to lock- and CAS-based designs.

6.2 Broader Implications

Beyond the specific technical contributions, this dissertation establishes several broader principles for designing concurrent and distributed systems.

6.2.1 Incremental Refinement Over All-or-Nothing Design

A recurring theme across all three projects is the rejection of all-or-nothing design choices. Traditional approaches force designers to choose up-front between ease-of-use and performance, between generality and specialization, between strong correctness guarantees and low overhead. This dissertation shows that well-designed and decoupled abstractions enable incremental refinement: programmers can start with simple, correct implementations and selectively optimize performance-critical components without overhauling entire systems.

In exoTM/STMCAS, this manifests as the ability to transform specific transactions from STM to STMCAS while leaving others unchanged. In Harmony, programmers can add new data structures or operations without reconsidering existing conflict detection logic. In RDMASTM, developers can adapt to workload characteristics by selecting caching granularity (field-level vs object-level), choosing clock implementations (centralized vs distributed), and configuring batching strategies—all without touching data structure code.

6.2.2 Stable Contracts Enable Independent Evolution

A key enabler of architectural decoupling is defining clear interface contracts between layers that expose sufficient semantic information for safe optimization without leaking implementation details. When interfaces capture essential requirements (e.g., conflict detection rules via orec semantics, version ordering/clock semantics, and ordering/visibility requirements for remote writes and reads) without prescribing implementation, different components can evolve independently.

In all three systems, we identified the right-sized interface required at each layer boundary. For exoTM, this meant exposing orec values and clock timestamps in a type-safe API, letting STMCAS use them as proofs for snapshotting and operation splitting while maintaining correctness guarantees. For Harmony, this meant separating the contract for data structure synchronization (temporal ownership and versioning) from transaction management (transactional ownership and conflict detection). For RDMASTM, this meant defining a stable high-level API (transactional field operations) separately from pluggable low-level services (orec management, caching, clocking).

These stable contracts have practical benefits:

- Adding new strategies (e.g., a cache or clock strategy) does not break existing code
- Optimizations target the appropriate layer without system-wide rewrites
- **Reasoning about correctness becomes tractable because each layer's responsibilities are scoped**

The last point is particularly valuable: by separating concerns, we can verify

properties at each layer independently rather than reasoning about the entire system monolithically.

6.2.3 Performance Heterogeneity Demands Policy Flexibility

The RDMASTM evaluation revealed an important insight: different policy dimensions have drastically different performance impacts. Cache policies provide nearly 2× improvements via 50–80% fewer RDMA operations; clock strategies differ within <20%; and batching ranges from about 5–10% (concentrated memory), and is often within ±2–3% when memory is widely distributed. This heterogeneity suggests that exposing all decisions as first-class configurable options, even those with modest impact, is valuable—it lets programmers focus optimization effort where it matters most.

More fundamentally, performance characteristics vary across workloads, data structures, and hardware platforms. What works well for read-heavy workloads may hurt write-heavy ones; what optimizes skip lists may harm hash tables; what helps on FDR InfiniBand may differ on Ethernet RDMA (RoCE) . Rather than attempting to identify a single optimal configuration, layered architectural decoupling acknowledges this diversity and provides the tools for systematic exploration.

6.2.4 The Role of Modern Language Features

This work also demonstrates that a key benefit of architectural decoupling is the ability to independently evolve policy implementations using modern language features. Harmony's use of C++ coroutines to manage transactional cleanup without explicit undo/redo logs shows how language-level abstractions can make certain

separations natural and efficient without altering the underlying synchronization mechanisms. Similarly, RDMASTM's use of Boost.Fiber for lightweight concurrency demonstrates that user-level threading can hide RDMA latency without kernel overhead, reducing context switch costs by approximately 1000× compared to OS threads.

These examples suggest that as programming languages evolve, new opportunities for architectural decoupling may emerge. Template metaprogramming, concepts, and compile-time policy composition could further reduce the runtime costs of flexibility. Complementary approaches—dynamic policy selection based on runtime profiling and profile-guided optimization—could automatically adapt strategies to workload phases.

6.3 Limitations and Future Directions

While this dissertation establishes architectural decoupling as a viable principle, several limitations and opportunities for future work remain.

6.3.1 Policy Discovery and Automated Tuning

The current systems require programmers to manually select policies based on workload characteristics and performance requirements. While our evaluations provide guidance (e.g., "cache policies matter most in RDMASTM"), systematic policy selection remains a manual process. Future work could explore automated policy discovery through profiling, machine learning-based policy selection, or adaptive systems that dynamically adjust policies based on runtime behavior.

The RDMASTM evaluation revealed that optimal thread and fiber configurations

vary across data structures and workloads with no consistent pattern. An interesting research direction would be developing heuristics or adaptive algorithms that automatically tune these parameters, potentially using reinforcement learning to explore the configuration space efficiently.

6.3.2 Verification and Correctness Tools

While we provide correctness arguments and, in some cases, proof sketches (e.g., Harmony's serializability arguments) , verifying that custom policies maintain system-wide invariants remains challenging. Future work could develop verification tools that check whether new policies satisfy the contracts required by the mechanism layer, perhaps using techniques from formal verification or model checking.

Similarly, debugging concurrent systems remains difficult. Tools that leverage architectural decoupling—for example, by logging policy decisions independently of mechanism operations—could improve debuggability and help programmers understand why certain behaviors emerge.

6.3.3 Extending to Additional Domains

This dissertation explored three domains: shared-memory concurrency, composable transactions, and RDMA-based distribution. The principle of architectural decoupling likely applies to other domains as well.

Persistent memory. Non-volatile memory technologies introduce new concerns (crash consistency, persist ordering) that could benefit from separating persistence mechanisms from usage policies. Building on exoTM's abstraction, one could envision pluggable policies for write-ahead logging, shadow paging, or copy-on-write

persistence.

Heterogeneous computing. GPU-accelerated data structures, FPGA-based synchronization, or CXL-attached memory introduce device-specific performance characteristics. Mechanism-policy separation could enable data structure implementations that work across devices while allowing device-specific optimization policies.

Tiered storage. Modern systems often span multiple storage tiers (DRAM, NVM, SSD, HDD). Separating data structure traversal mechanisms from data placement and migration policies could enable transparent tier-aware optimization.

Disaggregated architectures. Emerging datacenter designs disaggregate compute, memory, and storage. The RDMASTM approach could extend to these environments, where network characteristics differ from traditional RDMA clusters. (Initial exploration of different QP configurations in RDMASTM provides a starting point for this adaptation.)

6.3.4 Broader Policy Dimensions

While this work explored several policy dimensions (STM variants, cache strategies, clock mechanisms, orec placement), additional dimensions may prove valuable.

Memory reclamation policies. This work used validation-based reclamation (RDMASTM) and coroutine-based cleanup (Harmony), but other approaches (hazard pointers, reference counting) could be exposed as pluggable policies alongside the epoch-based reclamation already used in STMCAS.

Conflict detection granularity. Current systems use fixed-granularity conflict detection (field-level or object-level). Supporting multiple granularities simultaneously—coarse-grained for hot metadata, fine-grained for large objects—

could improve performance.

Contention management. When conflicts occur, systems must decide which transaction to abort. Exposing contention management as a pluggable policy (exponential backoff, priority-based, timestamp-based) could help optimize for specific fairness or latency goals.

6.3.5 Integration with Compilers and Runtimes

The exoTM work demonstrated compiler-compatible STM (xSTM), but deeper compiler integration could further reduce overheads. Compiler support for automatic policy selection based on static analysis, specialized code generation for different policies, or profile-guided policy optimization could improve performance and programmability.

Similarly, language runtime integration could enable more sophisticated policy composition. For example, a runtime could maintain per-thread policy contexts, allowing different application phases to use different policies, or dynamically adapt policies based on observed contention patterns.

It would also be interesting to explore how architectural decoupling principles fit into languages with strong static safety guarantees (e.g., Rust), where ownership and borrowing rules could statically verify layer contracts.

6.4 Closing Remarks

The challenges that motivated this dissertation—the difficulty of implementing correct, efficient concurrent data structures; the tension between ease-of-use and performance; the complexity of adapting synchronization mechanisms to new

environments—remain fundamental to parallel and distributed computing. As systems continue to grow in scale and complexity, the need for principled approaches to managing this complexity becomes ever more pressing.

This dissertation argues that architectural decoupling offers a path forward. By identifying stable abstractions that isolate low-level mechanisms from high-level policies, we can build systems that are simultaneously simple (correct by default, easy to reason about) and sophisticated (open to targeted optimization, adaptable to diverse contexts). The three projects presented here—exoTM/STMCAS, Harmony, and RDMASTM—demonstrate that this principle applies across shared memory, composable operations, and distributed systems, achieving both programmability and performance without requiring all-or-nothing compromises.

The broader lesson is one of humility and pragmatism. We should not expect to discover single optimal solutions for concurrent programming. Performance characteristics vary too much across workloads, hardware, and application requirements. Instead, we should build systems with clear contracts and well-defined extension points, enabling programmers to compose solutions from reusable components and apply domain knowledge where it matters most.

Concurrent programming will always be challenging. But by separating what must be correct (low-level synchronization mechanisms) from what must be efficient (programmer-facing abstractions and optimization strategies) , we can make it more manageable, more reliable, and more adaptable to the ever-changing landscape of modern computing.

Bibliography

[1] EW DIJKSTRA. Cooperating sequential processes (ewd123). *Programming Languages*, pages 43–112, 1965.

[2] Edsger W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, 1965.

[3] E.W̃. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.

[4] Tim Harris. A Pragmatic Implementation of Non-Blocking Linked Lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, Lisbon, Portugal, October 2001.

[5] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch Hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing*, Arcachon, France, September 2008.

[6] Maurice Herlihy, Yossi Lev, and Nir Shavit. A lock-free concurrent skiplist with wait-free search, 2007. URL `http://java.sun.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListMap.html`.

[7] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–268, Bangalore, India, January 2010.

[8] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The Art of Multiprocessor Programming, 2nd edition*. Morgan Kaufmann, 2021.

[9] Maurice P. Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[10] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, Ottawa, ON, Canada, August 1995.

[11] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proceedings of the 10th ACM Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[12] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[13] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. Transactional Data Structure Libraries. In *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation*, Santa Barbara, CA, June 2016.

[14] Kunal Agrawal, Angelina I-Ting Lee, and Jim Sukha. Safe Open-Nested Transactions Using Ownership. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, February 2009.

[15] Trevor Brown, William Sigouin, and Dan Alistarh. PathCAS: An Efficient Middle Ground for Concurrent Search Data Structures. In *Proceedings of the 27th ACM Symposium on Principles and Practice of Parallel Programming*, Seoul, South Korea, February 2022.

[16] Per Brinch Hansen. The Nucleus of a Multiprogramming System. *Communications of the ACM*, 13(4):238–241, April 1970.

[17] William Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, C. Pierson, and Fred Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–345, jun 1974.

[18] Roy Levin, Ellis Cohen, William Corwin, Fred Pollack, and William Wulf. Policy/Mechanism Separation in Hydra. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, Austin, TX, November 1975.

[19] Michael Spear Yaodong Sheng, Ahmed Hassan. Separating Mechanism from Policy in STM. In *Proceedings of the 32rd International Conference on Parallel Architectures and Compilation Techniques*, Vienna, Austria, October 2023.

[20] Michael Spear Yaodong Sheng, Ahmed Hassan. Transactional Data Structures with Orthogonal Metadata. In *Proceedings of the 30th ACM SIGPLAN Annual*

Symposium on Principles and Practice of Parallel Programming, Las Vegas, NV, February 2025.

[21] Yaodong Sheng, Ahmed Hassan, and Micheal Spear. Semantic Conflict Detection for Transactional Data Structure Libraries. In *Proceedings of the 33nd ACM Symposium on Parallelism in Algorithms and Architectures*, Philadelphia, USA, 2021.

[22] R. Kent Treiber. Systems Programming: Coping With Parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[23] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, May 1996.

[24] P. Lehman and S.B̃ing Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, December 1981.

[25] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[26] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[27] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66–76, 1996.

[28] Jeremy Manson, William Pugh, and Sarita Adve. The Java Memory Model. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, Long Beach, CA, January 2005.

[29] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ Concurrency. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, Austin, TX, January 2011.

[30] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28 (9):241–248, September 1979.

[31] Christoph Lameter. Effective Synchronization on Linux/NUMA Systems. In *Proceedings of the May 2005 Gelato Federation Meeting*, San Jose, CA, May 2005.

[32] Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[33] Eric H Jensen, Gary W Hagensen, and Jeffrey M Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical report, Technical Report UCRL-97663, Lawrence Livermore National Laboratory, 1987.

[34] James Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, Ottawa, ON, Canada, August 1995.

[35] Tim Harris, Keir Fraser, and Ian Pratt. A Practical Multi-word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing*, Toulouse, France, October 2002.

[36] Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, June 2003.

[37] Simon Doherty, David Detlefs, Lindsay Groves, Christine Flood, Victor Luchangco, Paul Martin, Mark Moir, Nir Shavit, and Guy Steele. DCAS is not a Silver Bullet for Nonblocking Algorithm Design. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures*, Barcelona, Spain, June 2004.

[38] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic Primitives for Non-blocking Data Structures. In *Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, July 2013.

[39] Shahar Timnat, Maurice Herlihy, and Erez Petrank. A Practical Transactional Memory Interface. In *Proceedings of the 2015 Euro-Par Conference*, Vienna, Austria, August 2015.

[40] Rachid Guerraoui, Alex Kogan, Virendra Marathe, and Igor Zablotchi. Efficient Multi-word Compare and Swap. In *Proceedings of the 34th International Symposium on Distributed Computing*, Virtual Event, October 2020.

[41] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, February 2008.

[42] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, September 2006.

[43] Torvald Riegel, Pascal Felber, and Christof Fetzer. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 221–228. ACM, 2007. doi: 10.1145/1248377.1248416.

[44] Intel. *Intel TSX Asynchronous Abort (TAA) - Technical Documentation*, 2019. Describes TSX mitigation via MSR IA32_TSX_CTRL (RTM_DISABLE, CPUID_CLEAR) and microcode support.

[45] Trevor Brown and Srivatsan Ravi. On the cost of concurrency in hybrid transactional memory. *arXiv preprint arXiv:1907.02669*, 2019.

[46] Mellanox Technologies (NVIDIA Networking). *RDMA Aware Networks Programming User Manual*, 2011. Verbs API overview, QPs, CQs, inline sends, etc.

[47] Infiniband verbs programming guide. `https://www.kernel.org/doc/Documentation/infiniband/`. Accessed 2025-09-25.

[48] Tom Shanley. *InfiniBand Network Architecture*. Addison-Wesley, 2003. URL `https://www.oreilly.com/library/view/infiniband-network-architecture/0201436641/`.

[49] Thomas Ziegler et al. Design guidelines for correct, efficient, and scalable synchronization with rdma. In *ACM SIGMOD*, 2023. URL `https://www.informatik.tu-darmstadt.de/media/systems/pdf_publications/SIGMOD_RDMA_Synchronization.pdf`.

[50] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *USENIX Annual Technical Conference (ATC)*, 2016. URL `https://www.pdl.cmu.edu/PDL-FTP/Storage/rdma_bench_atc.pdf`.

[51] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter rpcs can be general and fast. In *USENIX NSDI*, 2019. URL `https://www.usenix.org/system/files/nsdi19-kalia.pdf`.

[52] Aleksandar Dragojevic, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, Seattle, WA, April 2014.

[53] Xiaoqi Kong, Chunjie Lu, Xiangyao Zhou, et al. Understanding rdma microarchitecture resources for high throughput and low latency. In *USENIX NSDI*, 2023. URL `https://www.usenix.org/system/files/nsdi23-kong.pdf`.

[54] *RDMA Aware Networks Programming User Manual*. Mellanox Technologies, 2017. URL https://indico.cern.ch/event/218156/attachments/351725/490089/RDMA_Aware_Programming_user_manual.pdf. Rev. 1.3.

[55] On-demand paging (odp) overview. NVIDIA Networking Docs, 2025. URL https://docs.nvidia.com/networking/display/rdmacore50/on-demand+paging+odp.

[56] Amanda Baran, Jacob Nelson-Slivon, Lewis Tseng, and Roberto Palmieri. Alock: Asymmetric lock primitive for rdma systems. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 15–26, 2024.

[57] Ravi Rajwar and James R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2002.

[58] Amitabha Roy, Steven Hand, and Tim Harris. A Runtime System for Software Lock Elision. In *Proceedings of the EuroSys2009 Conference*, Nuremberg, Germany, March 2009.

[59] Maurice P. Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, Boston, MA, July 2003.

[60] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, King's College, University of Cambridge, September 2003.

[61] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, February 2008.

[62] Michael Spear, Maged M. Michael, and Christoph von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[63] Luke Dalessandro, Michael Spear, and Michael L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, January 2010.

[64] Tim Harris, Mark Plesko, Avraham Shinar, and David Tarditi. Optimizing Memory Transactions. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*, Ottawa, ON, Canada, June 2006.

[65] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, New York, NY, March 2006.

[66] Peng Wu, Maged M. Michael, Christoph von Praun, Takuya Nakaike, Rajesh Bordawekar, Harold W. Cain, Calin Cascaval, Siddhartha Chatterjee, Stefanie Chiras, Rui Hou, Mark Mergen, Xiaowei Shen, Michael Spear, Hua Yong Wang, and Kun Wang. Compiler and Runtime Techniques for Software Transactional Memory Optimization. *Concurrency and Computation: Practice & Experience*, 21(1):7–23, January 2009.

[67] Minjia Zhang, Jipeng Huang, Man Cao, and Michael Bond. Low-Overhead Software Transactional Memory with Progress Guarantees and Strong Semantics. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*, San Francisco, CA, February 2015.

[68] Takayuki Usui, Yannis Smaragdakis, Reimer Behrends, and Jacob Evans. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *Proceedings of the 18th International Conference on Parallel Architecture and Compilation Techniques*, Raleigh, NC, September 2009.

[69] Chao Wang, Yujie Liu, and Michael Spear. Transaction-Friendly Condition Variables. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, Prague, Czech Republic, June 2014.

[70] Brian Demsky and Navid Tehrany. Integrating File Operations into Transactional Memory. *Journal of Parallel and Distributed Computing*, 71(10): 1293–1304, 2011.

[71] William N. Scherer III and Michael L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[72] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing*, Denver, CO, August 2006.

[73] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic Contention Management in SXM. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, September 2005.

[74] Tomer Heber, Danny Hendler, and Adi Suissa. On the Impact of Serializing Contention Management on STM Performance. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, December 2009.

[75] Luke Dalessandro, Michael L. Scott, and Michael Spear. Transactions as the Foundation of a Memory Consistency Model. In *Proceedings of the 24th International Symposium on Distributed Computing*, Cambridge, MA, September 2010.

[76] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[77] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, USA, October 2008.

[78] ISO/IEC JTC 1/SC 22/WG 21. Technical Specification for C++ Extensions for Transactional Memory, May 2015. URL `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf`.

[79] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, UT, March 2014.

[80] Aleksandar Dragojevic and Tim Harris. STM in the Small: Trading Generality for Performance in Software Transactional Memory. In *Proceedings of the EuroSys2012 Conference*, Bern, Switzerland, April 2012.

[81] Robert Ennals. Software Transactional Memory Should Not Be Lock Free. Technical Report IRC-TR-06-052, Intel Research Cambridge, 2006.

[82] David Dice and Nir Shavit. TLRW: Return of the Read-Write Lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[83] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, September 2007.

[84] Dawson Engler, Frans Kaashoek, and O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, December 1995.

[85] Donald Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, March 2011.

[86] Richard Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[87] Justin Gottschlich, Manish Vachharajani, and Jeremy Siek. An Efficient Software Transactional Memory Using Commit-Time Invalidation. In *Proceedings of the 2010 International Symposium on Code Generation and Optimization*, Toronto, ON, Canada, April 2010.

[88] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, September 2005.

[89] Virendra Marathe and Mark Moir. Toward High Performance Nonblocking Software Transactional Memory . In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, February 2008.

[90] Wenjia Ruan, Yujie Liu, and Michael Spear. Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters. *ACM Transactions on Architecture and Code Optimization*, 10(4):40:1–40:21, December 2013.

[91] Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, September 2006.

[92] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[93] Michael Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, September 2008.

[94] David Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, March 2009.

[95] Luke Dalessandro and Michael L. Scott. Sandboxing Transactional Memory. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, September 2012.

[96] Michael Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, February 2009.

[97] Richard Yoo and Hsien-Hsin Lee. Adaptive Transaction Scheduling for Transactional Memory Systems. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[98] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of Memory Reclamation for Lockless Synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[99] Trevor Brown. Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way. In *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing*, Portland, OR, June 2015.

[100] Nachshon Cohen and Erez Petrank. Automatic Memory Reclamation for Lock-Free Data Structures. *ACM SIGPLAN Notices*, 50(10):260–279, 2015.

[101] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, Asilomar State Beach, CA, July 2016.

[102] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. Interval-Based Memory Reclamation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Vienna, Austria, February 2018.

[103] Ajay Singh, Trevor Brown, and Ali Mashtizadeh. NBR: Neutralization Based Reclamation. In *Proceedings of the 26th ACM Symposium on Principles and Practice of Parallel Programming*, Seoul, South Korea, February 2021.

[104] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. Forkscan: Conservative Memory Reclamation for Modern Operating Systems. In *Proceedings of the EuroSys2017 Conference*, Belgrade, Serbia, April 2017.

[105] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. ThreadScan: Automatic, Scalable Memory Reclamation. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, Portland, OR, June 2015.

[106] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the EuroSys2018 Conference*, Porto, Portugal, April 2018.

[107] Hans-Juergen Boehm. Can Seqlocks Get Along with Programming Language Memory Models? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 12–20, Beijing, China, June 2012.

[108] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William Scherer, and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. In *Proceedings of the 9th international conference on Principles of Distributed Systems*, Pisa, Italy, December 2006.

[109] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul, Turkey, March 2015.

[110] Free Software Foundation. Transactional Memory in GCC, 2012. http://gcc.gnu.org/wiki/TransactionalMemory.

[111] Pantea Zardoshti, Tingzhe Zhou, Pavithra Balaji, Michael Scott, and Michael Spear. Simplifying Transactional Memory Support in C++. *ACM Transactions on Architecture and Code Optimization*, 16(3):25:1–25:24, July 2019.

[112] Dave Dice and Nir Shavit. Understanding Tradeoffs in Software Transactional Memory. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, San Jose, CA, March 2007.

[113] Luke Dalessandro, Virendra Marathe, Michael Spear, and Michael Scott. Capabilities and Limitations of Library-Based Software Transactional Memory in C++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, August 2007.

[114] Trevor Brown. A template for implementing fast lock-free trees using htm. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 293–302, 2017.

[115] Vincent Gramoli. More Than You Ever Wanted to Know about Synchronization. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*, San Francisco, CA, February 2015.

[116] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, Farminton, Pennsylvania, November 2013.

[117] Philip Lehman and Hsiang-Tsung Kung. Concurrent Manipulation of Binary Search Trees. *ACM Transactions on Database Systems*, 5(3):354–382, 1980.

[118] Al Danial. Count Lines of Code, 2006–2022. https://github.com/AlDanial/cloc/.

[119] Yujie Liu, Kunlong Zhang, and Michael Spear. Dynamic-Sized Nonblocking Hash Tables. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing*, Paris, France, July 2014.

[120] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic Transactions. *Journal of Parallel and Distributed Computing*, 100:103–127, February 2017.

[121] Yehuda Afek, Hillel Avni, and Nir Shavit. Towards Consistency Oblivious Programming. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, Toulouse, France, December 2011.

[122] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Optimistic Transactional Boosting. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming*, Orlando, FL, February 2014.

[123] Nachshon Cohen, Maurice Herlihy, Erez Petrank, and Elias Wald. The Teleportation Design Pattern for Hardware Transactional Memory. In *Proceedings of the 21st International Conference on Principles of Distributed Systems*, Lisbon, Portugal, December 2017.

[124] Tingzhe Zhou, Victor Luchangco, and Michael Spear. Hand-Over-Hand Transactions with Precise Memory Reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, Washington, DC, July 2017.

[125] Lingxiang Xiang and Michael L. Scott. Software Partitioning of Hardware Transactions. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*, San Francisco, CA, February 2015.

[126] Yossi Lev and Jan-Willem Maessen. Split Hardware Transactions: True Nesting of Transactions Using Best-Effort Hardware Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, February 2008.

[127] Naama Ben-David, Guy Blelloch, and Yuanhao Wei. Lock-Free Locks Revisited. In *Proceedings of the 27th ACM Symposium on Principles and Practice of Parallel Programming*, Seoul, South Korea, February 2022.

[128] Rachid Guerraoui and Vasileios Trigonakis. Optimistic Concurrency with OPTIK. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming*, February 2016.

[129] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, 1976.

[130] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the JACM*, 26(4):631–653, 1979.

[131] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware transactions for faster concurrent code. In Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues, editors, *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 31:1–31:16. ACM, 2016.

[132] Deli Zhang, Pierre Laborde, Lance Lebanoff, and Damian Dechev. Lock-Free Transactional Transformation for Linked Data Structures. *ACM Transactions on Parallel Computing*, 5(1), June 2018.

[133] Wentao Cai, Haosen Wen, and Michael L Scott. Transactional composition of nonblocking data structures. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 441–443, 2023.

[134] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, February 2008.

[135] Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Brief announcement: Proust: A design space for highly-concurrent transactional data structures. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 251–253. ACM, 2017.

[136] Thomas Dickerson, Eric Koskinen, Paul Gazzillo, and Maurice Herlihy. Conflict Abstractions and Shadow Speculation for Optimistic Transactional Objects. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, Bali, Indonesia, November 2019.

[137] Amos Israeli and Lihu Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, 1994.

[138] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2003.

[139] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Seattle, WA, September 2008.

[140] Ferad Zyulkyarov, Vladimir Gajinov, Osman Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, February 2009.

[141] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. Improving In-Memory

Database Index Performance with Intel Transactional Synchronization Extensions. In *Proceedings of the 20th International Symposium on High-Performance Computer Architecture*, Orlando, FL, February 2014.

[142] Qingsen Wang, Pengfei Su, Milind Chabbi, and Xu Liu. Lightweight Hardware Transactional Memory Profiling. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, Washington, DC, February 2019.

[143] Eliot Moss. Open Nested Transactions: Semantics and Support. In *Proceedings of the 4th Workshop on Memory Performance Issues*, Austin, TX, February 2006.

[144] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. OneFile: A Wait-Free Persistent Transactional Memory. In *Proceedings of the 49th International Conference on Dependable Systems and Networks*, Portland, OR, June 2019.

[145] Maya Arbel-Raviv and Trevor Brown. Harnessing Epoch-Based Reclamation for Efficient Range Queries. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Vienna, Austria, February 2018.

[146] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 168–183. ACM, 2015.

[147] Yuanhao Wei, Naama Ben-David, Guy Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time Snapshots with Applications to Concurrent Data Structures. In *Proceedings of the 26th ACM Symposium on Principles and Practice of Parallel Programming*, Seoul, South Korea, February 2021.

[148] Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. Bundling linked data structures for linearizable range queries. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 368–384, 2022.

[149] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[150] Michael Scott. *Programming Language Pragmatics, 4th edition.* Morgan Kaufmann, 2016.

[151] Jason Evans. jemalloc memory allocator, 2017. http://http://jemalloc.net/.

[152] Maged Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, Winnipeg, Manitoba, Canada, August 2002.

[153] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. Understanding and Optimizing Persistent Memory Allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, June 2020.

[154] Pedro Ramalhete and Andreia Correia. Scaling Up Transactions with Slower Clocks. In *Proceedings of the 29th ACM Symposium on Principles and Practice of Parallel Programming*, Edinburgh, UK, March 2024.

[155] 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)*, pages 87–104. ACM, 2015. doi: 10.1145/2815400.2815419.

[156] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services (herd). In *Proceedings of the ACM SIGCOMM Conference*, pages 295–306, 2014. doi: 10.1145/2619239.2626299. URL `https://www.cs.cmu.edu/~xia/resources/Documents/herd-sigcomm2014-readable.pdf`.

[157] 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*, pages 1033–1048, 2022.

[158] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 185–201, 2016. URL `https://www.usenix.org/system/files/conference/osdi16/osdi16-kalia.pdf`.

[159] NVIDIA Networking. Rdma aware networks programming user manual. `https://docs.nvidia.com/networking/display/RDMAAwareProgrammingv17`, 2023. Rev. 1.7; covers one-sided verbs and atomics (CAS, FAA).

[160] InfiniBand Trade Association. Infiniband architecture specification, volume 1, release 1.2.1. Technical report, IBTA, 2007. URL `https://www.afs.enea.it/asantoro/V1r1_2_1.Release_12062007.pdf`.

[161] 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 USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019. URL `https://www.flux.utah.edu/paper/duplyakin-atc19`.

[162] Sanidhya Kashyap, Dai Qin, Steve Byan, Virendra J Marathe, and Sanketh Nalli. Correct, fast remote persistence. *arXiv preprint arXiv:1909.02092*, 2019.

[163] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. Characterizing and optimizing remote persistent memory with {RDMA} and {NVM}. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 523–536, 2021.

[164] Rong Chen, Jinyang Wang, Zhi Yang, Binyu Zang, and Haibo Chen. Drtm+h: Eluding doomed hdtm transactions by petrifying concurrency control. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 601–618, 2018. URL `https://www.usenix.org/system/files/osdi18-chen-rong.pdf`.

[165] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Donald Kossmann. Rethinking database architectures for high-performance rdma networks. In *Proceedings of the 13th International Workshop on Data Management on New Hardware (DaMoN)*, pages 1–10, 2017. URL `https://www.meltem.kaust.edu.sa/sites/default/files/2019-09/zamanian-damon17.pdf`.

[166] Aleksandar Dragojević, Rachid Guerraoui, and Michał Kapałka. Stretching transactional memory (swisstm). In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 155–165, 2009. doi: 10.1145/1542476.1542494. URL `https://kapalka.eu/files/stretching-tm-pldi09.pdf`.

[167] Walid Reda, Prashanth R., Danyang Zhuo, Srinivas Narayana, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Rdma is turing complete, we just did not know it yet! In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022. URL `https://www.usenix.org/system/files/nsdi22spring_prepub_reda.pdf`.

[168] Jianbo Wu, Jie Liu, Gokcen Kestor, Roberto Gioiosa, Dong Li, and Andres Marquez. Performance study of cxl memory topology. In *Proceedings of the International Symposium on Memory Systems*, pages 172–177, 2024.

Vita

Yaodong Sheng

Education:

- Ph.D. in Computer Science, Lehigh University, 2025
- B.S. in Computer Science, Beijing University of Post and Telecommunications, 2016

Publications:

- Yaodong Sheng and Michael Spear. "Transactional Data Structures with Orthogonal Metadata." *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2025.
- Yaodong Sheng and Michael Spear. "Separating Mechanism from Policy in STM." *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2023.
- Yaodong Sheng, Ahmed Hassan, and Michael Spear. "Semantic Conflict Detection for Transactional Data Structure Libraries." *Proceedings of the*

ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2021.

Professional Experience:

- Research Assistant, Lehigh University, [2019–2025]
- Teaching Assistant, Lehigh University, [2022, 2023]