WriteBoost RCU: an Enhanced RCU Library that provides an RCU-centric Update-side Synchronization Mechanism

Jiawei He
jiaweihe@umich.edu

Jirong Yang
yjrcs@umich.edu

Fangyi Dai
fdai@umich.edu

Ziang Wang
ziangw@umich.edu

Abstract

Read-Copy Update (RCU) is a synchronization mechanism widely used in the Linux kernel that has been adapted for user-space applications through libraries like liburcu and Folly RCU. While these implementations excel at providing wait-free reads, they delegate update synchronization to users, leading to increased complexity and potentially suboptimal performance. This paper presents WriteBoost RCU, an enhanced RCU library that integrates update-side synchronization responsibilities while maintaining RCU's exceptional read performance.

Our implementation introduces several key innovations: a single updater/reclaimer design that eliminates synchronization overhead in retirement lists, batch processing of updates to amortize copy costs, and an object pool to optimize memory management. Through comprehensive benchmarking, we demonstrate that WriteBoost RCU not only simplifies the programming model but also achieves superior performance across various scenarios. Notably, our evaluation challenges the traditional view that RCU is primarily beneficial for read-heavy workloads, showing competitive write performance even in balanced read-write scenarios.

The results indicate that RCU's potential applications may be broader than previously thought, particularly when enhanced with efficient update-side synchronization. WriteBoost RCU demonstrates that RCU libraries can successfully optimize write operations without compromising their fundamental benefits, suggesting new directions for future RCU implementations.

1 Introduction

Read-Copy Update (RCU) is a synchronization mechanism that has been widely used in the Linux kernel for over two decades, providing excellent read-side performance for concurrent data structures. Initially designed for kernel-space usage, RCU has found its place in user-space applications through libraries like liburcu [1], which provides a user-space implementation of RCU under the LGPLv2.1 license. The significance of RCU in modern concurrent programming is further evidenced by its upcoming inclusion in C++26 [4] and its production-ready implementation in Folly [2], Facebook/Meta's open-source C++ library.

Traditional RCU libraries primarily function as deferred memory reclamation mechanisms. While writers typically perform both update and reclamation operations, conceptually separating these operations helps understand the limitations of existing RCU implementations. As shown in Table 1, current RCU libraries only provide synchronization between read operations and reclaim operations, leaving update synchronization as a responsibility of the user.

This division of responsibilities exposes users to significant complexity when implementing updater code, as illustrated in Listing 1. Users must:

- Manually manage atomic pointers with appropriate memory ordering for correctness and performance
- Implement the read-copy-update operation themselves
- Employ additional synchronization mechanisms (e.g., mutexes) to maintain consistency and prevent update conflicts

These requirements lead to poor update-side programmability and potentially suboptimal performance, as users must implement their own synchronization strategies without the benefit of RCU-specific optimizations.

Our proposed solution addresses these limitations by extending the RCU library to handle update synchronization. This enhanced RCU library manages synchronization across all three operation types: read, write, and reclaim (Table 1). This approach offers several advantages:

1. **Improved Usability**: By encapsulating all synchronization within the library, we significantly reduce the complexity faced by users.
2. **RCU-Centric Update Synchronization**: Taking responsibility for update synchronization allows the library to implement optimizations specifically designed around RCU's characteristics.

```
extern std::atomic<Data*> ptr;
extern std::mutex mut;
void update() {
  mut.lock();
  // updaters synchronize through mutex,
  // so we only need relaxed ordering here
  auto newPtr =
    new Data{*ptr.load(std::memory_order_relaxed)};
  doSomeUpdate(newPtr);
  // release store so that the update
  // is visible to readers
  auto oldPtr = ptr.exchange(
    newPtr, std::memory_order_release);
  mut.unlock();
  // Existing RCU library only provides
  // deferred reclamation service
  rcu_retire(oldPtr);
}
```

Listing 1: Simple Concurrent Updater code using existing RCU libraries

Table 1: Synchronization Responsibility of RCU libraries vs. user

	Existing RCU libraries	WriteBoost RCU
Library	Read, Reclaim	Read, Update, Reclaim
User	Update	None

3. **Wider Applicability**: The combination of easier usage and better update-side performance makes RCU suitable for a broader range of concurrent programming scenarios.

The implementation of our enhanced RCU library is available as open source from https://github.com/waker-he/writeboost-rcu. CSE 582 Github classroom repository link: https://github.com/eecs582/fall2024-project-group-rcu.

2 Background and Related Work

2.1 Read-Copy-Update (RCU)

Read-Copy Update (RCU) is a synchronization mechanism that can be viewed as a scalable alternative to read/write locks. The fundamental idea of RCU is illustrated in Listing 2, where read operations maintain wait-free characteristics by simply dereferencing an atomic pointer. Writers follow a three-step process: first creating a copy of the data, then performing updates on this private copy, and finally publishing the update through an atomic exchange operation. However, this basic implementation faces a critical challenge: the old version of the object cannot be immediately deleted as concurrent

```
std::atomic<Data*> ptr;
void read() {
    doSomething(*ptr);
}
void write() {
    Data* copied = new Data(*ptr);    // Read-copy
    update(copied);                   // Update
    Data* oldPtr = ptr.exchange(copied);
    // leaking oldPtr
}
```

Listing 2: RCU without RCU libraries

```
std::atomic<Data*> ptr;
void read() {
    rcu_read_lock();
    // start of read-side critical section
    doSomething(*ptr);
    // end of read-side critical section
    rcu_read_unlock();
}
void write() {
    Data* copied = new Data(*ptr);    // Read-copy
    update(copied);                   // Update
    Data* oldPtr = ptr.exchange(copied);
    rcu_retire(oldPtr);
}
```

Listing 3: RCU with RCU library

readers might still be accessing it.

This memory reclamation challenge is addressed by RCU libraries, as shown in Listing 3. Readers explicitly mark their read-side critical sections using `rcu_read_lock()` and `rcu_read_unlock()`. Writers can then safely delegate the memory reclamation responsibility to the RCU library through `rcu_retire()`. The library ensures that the old version is only reclaimed after all readers that might be accessing it have exited their read-side critical sections.

RCU readers benefit from two key design choices:

1. **Wait-free Operation**: Readers are never blocked by writers or other readers, enabling true parallel reads.
2. **Thread-local Counter Design**: Unlike read/write locks (e.g., `std::shared_mutex`) that use a shared atomic counter and suffer from cache line contention, RCU libraries typically employ thread-local counters for each reader thread, eliminating this bottleneck.

These characteristics give RCU readers unparalleled scalability compared to other synchronization mechanisms.

However, RCU writers face several challenges. Compared to simple mutex-based synchronization, RCU writers must perform additional operations including memory allocation, object copying, and old version reclamation. Writers also face

increased programming complexity, as shown in Listing 1 (note that Listings 2 and 3 simplify the scenario by assuming a single writer thread, thus omitting mutex synchronization). Due to these trade-offs, RCU is traditionally recommended for read-heavy scenarios where reader performance is crucial [6].

2.2 Related Work

Two notable works have attempted to improve RCU writers' programmability and performance:

The Read-Log-Update (RLU) mechanism [3] provides a simplified programming model similar to Software Transactional Memory (STM), requiring only that developers explicitly mark RLU reader sections. RLU handles all synchronization automatically, eliminating the complex copy management and pointer manipulation required in RCU. However, RLU incurs non-negligible read-side overhead compared to RCU's wait-free reads, making it unsuitable for some read-heavy workloads.

RCU-HTM [5] takes an innovative approach by combining RCU with Hardware Transactional Memory (HTM) to implement high-performance concurrent binary search trees. The design allows RCU's wait-free reads while using HTM to enable multiple concurrent writers. However, RCU-HTM is specifically designed for binary search trees rather than being a general-purpose RCU library solution.

3 Key Design and Implementation

3.1 Interface Design

WriteBoost RCU provides a simple and intuitive interface through a class template `rcu_protected<T>`, where T is the type of object to protect. The interface design focuses on minimizing the complexity exposed to users while maintaining the high performance characteristics of RCU. A key distinction from existing RCU libraries is that WriteBoost RCU provides synchronization for both read-write and write-write synchronization, significantly simplifying the programming model.

As shown in Listing 4, the interface consists of two primary operations that encapsulate the complexity of RCU synchronization.

The reader interface provides the `get_ptr()` method which returns a `std::unique_ptr` with a custom deleter that automatically handles read-side critical section entry and exit. Internally, it calls `rcu_read_lock` to increment a thread-local read counter, and the custom deleter ensures `rcu_read_unlock` is called when the pointer goes out of scope. This RAII-style design eliminates common programming errors where developers might forget to mark the end of a read-side critical section.

The writer interface consists of a single `update()` method that takes a callback function. This method encapsulates all

```
rcu_protected<int> rp{new int(0)};

void reader() {
    auto ptr = rp.get_ptr();
    doSomething(*ptr);
}

void writer() {
    rp.update([](int* ptr) { doSomeUpdate(ptr); });
}
```

Listing 4: Basic usage of WriteBoost RCU interface

the complexity of managing object copies, synchronization, and memory reclamation. This is in stark contrast to traditional RCU where writers must manually handle copy creation, pointer manipulation, and memory reclamation timing. The update callback is guaranteed to execute atomically with respect to other update callbacks, providing proper synchronization between writers.

The interface design provides significant benefits in both simplicity and robustness. Writers and readers only need to call a single function to perform their operations, making the code much simpler compared to traditional RCU implementations. The interface also eliminates possibilities of data races by ensuring readers only have access through pointers to `const` returned by `get_ptr()`, and all updates must go through the synchronized `update()` function. When the reader's pointer goes out of scope, the custom deleter ensures proper cleanup, preventing any access after the read-side critical section ends.

3.2 RCU-Centric Update-Side Synchronization

The core innovation in WriteBoost RCU is its RCU-centric update-side synchronization mechanism, which addresses several fundamental limitations in traditional RCU libraries. This mechanism is built on three key components: a single updater/reclaimer design, batch updates processing, and an object pool for memory management.

3.2.1 Single Updater/Reclaimer

A key insight of WriteBoost RCU is to unify the roles of updater and reclaimer into a single thread. In traditional RCU implementations, multiple updaters can concurrently call `rcu_retire()`, requiring complex synchronization mechanisms to prevent race conditions in the retirement lists. WriteBoost RCU eliminates this overhead by ensuring only one thread acts as both updater and reclaimer at any time.

When a thread calls `update()`, it first attempts to register as the updater. If successful, this thread becomes responsible

for both performing updates and managing memory reclamation through the retirement lists. Other threads that need to perform updates simply enqueue their update callbacks to be processed by the current updater thread. This design ensures there is only one reclaimer at a time, eliminating the need for extra synchronization for the retire lists. The result is better reclamation performance compared to existing RCU libraries, which must implement complex synchronization to handle concurrent `rcu_retire` calls safely.

3.2.2 Batch Updates

WriteBoost RCU addresses a common performance issue in RCU implementations: the need to create a new copy for every update to avoid blocking readers. This frequent memory allocation and copy operation can incur significant overhead, leading to poor update-side performance for RCU clients.

Our solution performs batch updates on a single copy. When a thread becomes the updater, it processes not only its own update but also any pending updates in the queue on the same object copy. This approach amortizes the overhead across multiple updates, improving overall performance. Additionally, since there is only one updater thread processing a batch of updates, this approach reduces cache misses. Later updates in a batch operate on a hot cache, performing their operations more efficiently.

To prevent updates from being indefinitely delayed, the system implements a configurable flushing threshold that limits the maximum batch size. This ensures updates become visible to readers in a timely manner while maintaining the performance benefits of batching.

3.2.3 Object Pool

To further optimize memory utilization and reduce update overhead, WriteBoost RCU implements an object pool consisting of retired objects that are ready to be reclaimed (where no readers are accessing them anymore). During update operations, the implementation can reuse memory from this object pool, reducing the overhead of updates to a simple copy assignment operation.

The object pool is particularly effective when combined with batch updates since multiple updates can efficiently reuse the same memory locations. This approach not only reduces memory allocation overhead but also improves cache efficiency by maintaining a hot set of objects. The result is a significant reduction in the memory management overhead typically associated with RCU updates.

3.3 Epoch-Based Reader-Reclaimer Synchronization

WriteBoost RCU implements an epoch-based synchronization mechanism between readers and the reclaimer that ensures safe memory reclamation while maintaining RCU's excellent read-side performance. The core idea of the mechanism involves tracking reader activity through epochs and carefully managing object retirement and reclamation.

The implementation uses a sliding window of two epochs, represented by a simple boolean value where the previous epoch is the logical negation of the current epoch. Rather than using a conventional atomic integer for epoch tracking, this design choice simplifies the implementation while providing all necessary functionality. An important characteristic of this system is that each epoch can correspond to multiple versions of the protected object - a reader accessing any version from an epoch prevents the reclamation of all objects within that epoch.

Each epoch maintains its own list of retired objects waiting to be reclaimed. When an object is retired, it is added to the current epoch's list. The objects in each epoch's list are protected by different conditions: objects in the current epoch's list are protected by readers in the current epoch, while objects in the previous epoch's list are protected by readers in the previous epoch. When the reclaimer attempts to increment the current epoch (by flipping the boolean value), it must first ensure that all readers accessing objects from the previous epoch have finished. Once this condition is met, the reclaimer can safely clean up the list of retired objects corresponding to the previous epoch and reuse that list for the next epoch. If there are still readers accessing objects from the previous epoch, any newly retired object is appended to the current epoch's list of retired objects.

To achieve amortized constant-time complexity for reclamation operations, WriteBoost RCU optimizes when epoch advancement checks occur. Instead of checking if the current epoch can be incremented on every object retirement, the implementation only performs this check when the size of the current epoch's retired object list exceeds a threshold based on the number of reader threads. This means that for every P objects reclaimed (where P is the number of reader threads), there is only one scan of all thread-local counters (with time complexity O(P)), effectively achieving amortized O(1) time complexity for each object's reclamation.

The use of thread-local counters for tracking reader activity is a crucial aspect of this design. Each reader thread maintains its own counter, eliminating the contention typically found in reader-writer locks that use shared counters. This approach allows readers to proceed without any synchronization overhead while still enabling the reclaimer to safely determine when all readers from a particular epoch have completed their operations.

4 Evaluation

To assess the performance characteristics of WriteBoost RCU (WBRCU), we conducted comparative benchmarking against three other synchronization mechanisms: Folly's RCU im-

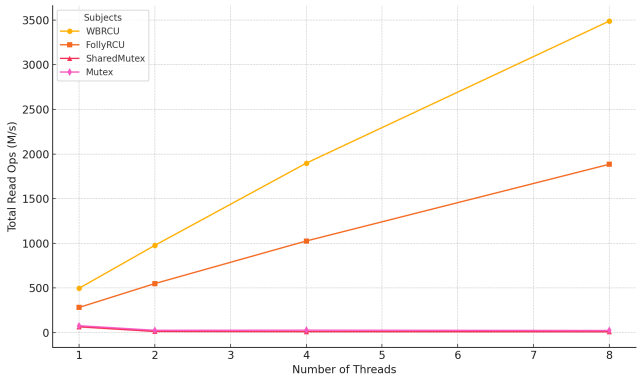

Figure 1: Read Throughput Scaling with Thread Count.

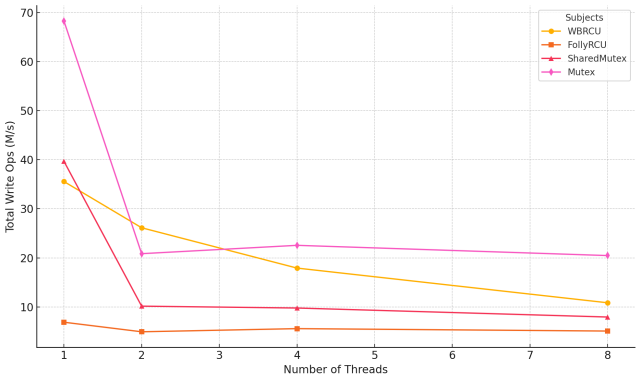

Figure 2: Write Throughput Scaling with Thread Count.

plementation, `std::shared_mutex` (read/write lock), and `std::mutex`. All benchmarks were executed on an 8-core 2400 MHz CPU.

Our evaluation framework considers four key parameters that influence concurrent data structure performance:

- Total number of threads
- Length of critical section
- Size of the protected data
- Read-write ratio

Given the vast parameter space created by these variables, exhaustively testing all combinations would be impractical. Instead, our methodology focuses on isolating the impact of each parameter by fixing the other three at controlled baseline values. This approach allows us to systematically analyze how each factor affects performance while maintaining experimental clarity.

4.1 Varying Total Number of Threads

To evaluate how different synchronization mechanisms scale with increasing thread counts, we first tested performance under pure read and pure write workloads. The protected data structure in these tests was a 64-bit integer, with writes performing simple increments and reads accessing the value directly. Figures 1 and 2 illustrate the throughput scaling from 1 to 8 threads for both operations.

For read operations, WriteBoost RCU demonstrates superior scalability and absolute performance compared to all other synchronization mechanisms. At a single thread, WBRCU achieves approximately 496 million reads per second, about 1.8x faster than Folly RCU and 7.6x faster than std::shared_mutex. This performance advantage grows with thread count - at 8 threads, WBRCU reaches 3.49 billion reads per second, maintaining its lead over Folly RCU (1.88 billion ops/s) while dramatically outperforming both mutex variants which show poor scaling due to contention. The near-linear scaling of WBRCU's read performance can be attributed to its efficient thread-local counter design and the complete elimination of synchronization overhead in the read path.

Write operation performance presents a more nuanced picture. At a single thread, std::mutex provides the highest throughput at 68.3 million writes per second, followed by std::shared_mutex at 39.7 million ops/s and WBRCU at 35.6 million ops/s. However, as thread count increases, all mechanisms show degraded performance due to contention, with WBRCU maintaining better throughput than Folly RCU across all thread counts. At 8 threads, WBRCU achieves 10.9 million writes per second, which is lower than std::mutex (20.5M ops/s) but higher than std::shared_mutex (7.9M ops/s) and significantly better than Folly RCU (5.1M ops/s).

The write performance characteristics reflect fundamental trade-offs in the different synchronization approaches. While traditional mutex-based approaches optimize for single-threaded write performance, WBRCU's additional overhead comes from its copy-on-write semantics and memory management. However, WBRCU's single updater/reclaimer design and batch processing help maintain reasonable write performance under contention while enabling its superior read-side scalability.

These results demonstrate that WriteBoost RCU successfully achieves its primary design goal of maintaining RCU's exceptional read-side performance while providing competitive write-side performance through its RCU-centric synchronization mechanism. The significant read-side advantage makes it particularly well-suited for read-heavy workloads where conventional RCU is typically employed.

4.2 Varying Length of Critical Section

To understand how different synchronization mechanisms handle varying workload intensities, we evaluated read performance with increasing critical section lengths, simulated through busy-waiting. The experiment used 8 concurrent

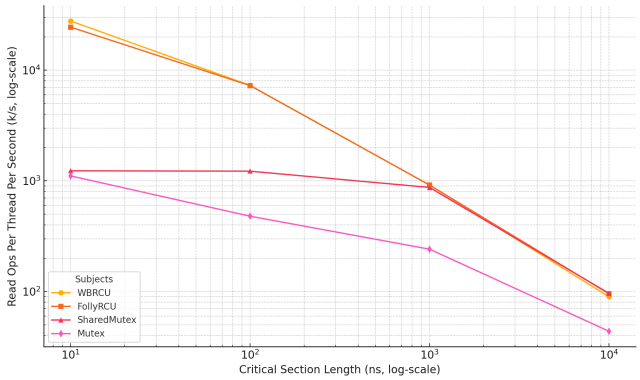

Figure 3: Read Operation Per Thread with Length of Critical Section Increased from 10ns to 10000ns (Log-scale)

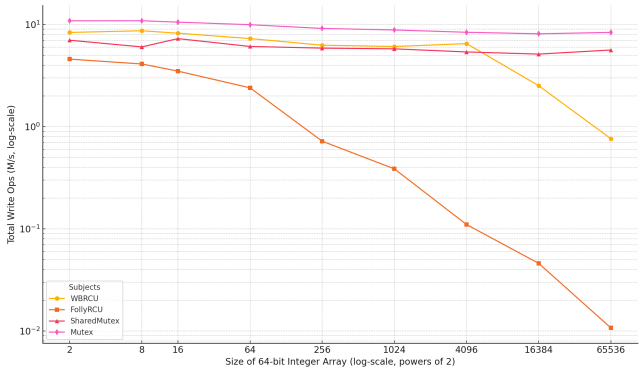

Figure 4: Total Write Operation with Increasing Size of Protected Data

reader threads and measured performance as critical section length increased from 10ns to 10000ns. Figure 3 illustrates these results using a log-log scale to capture the wide performance range.

At the shortest critical section length (10ns), both RCU implementations significantly outperform the mutex-based approaches. WBRCU achieves 27.8M ops/thread/second, slightly ahead of Folly RCU's 24.5M ops/thread/second, while std::shared_mutex and std::mutex lag significantly at 1.23M and 1.11M ops/thread/second respectively. This order-of-magnitude performance gap demonstrates RCU's fundamental advantage when synchronization overhead dominates execution time.

As critical section length increases, several notable trends emerge:

1. Both RCU implementations show nearly identical performance degradation curves, with throughput inversely proportional to critical section length. This is the expected optimal behavior, as the execution time becomes dominated by the critical section rather than synchronization overhead.

2. std::shared_mutex maintains relatively stable performance up to 100ns critical sections, suggesting its synchronization overhead masks the impact of shorter critical sections. Beyond 1000ns, it converges with the RCU implementations as the critical section length dominates total execution time.

3. std::mutex shows the poorest scaling, with performance degrading more rapidly than other approaches as critical section length increases. At 10000ns, it achieves only 43.8K ops/thread/second, less than half the throughput of other mechanisms.

The results validate RCU's design principle of minimizing read-side synchronization overhead. Even as critical sections grow longer, both RCU implementations maintain their performance advantage or match other synchronization mechanisms. WriteBoost RCU's slight performance edge over Folly RCU at shorter critical sections suggests its implementation introduces marginally less overhead in the read path.

This benchmark particularly highlights why RCU is an excellent choice for read-heavy workloads with short critical sections, where its negligible synchronization overhead provides maximum benefit. The performance convergence at longer critical sections also indicates that RCU remains a competitive choice even when critical sections are more substantial, as it never performs worse than traditional synchronization mechanisms.

4.3 Varying Size of the Protected Data

To evaluate how different synchronization mechanisms handle varying data sizes, we measured write performance while increasing the size of the protected data structure from 2 to 65,536 64-bit integers. This benchmark specifically focuses on write performance with 8 concurrent writer threads, as data size primarily impacts write operations due to the copy-on-write semantics of RCU implementations. Figure 4 illustrates these results using a log-log scale.

For small data sizes (2-16 integers), all synchronization mechanisms maintain relatively stable performance, with std::mutex achieving the highest throughput at approximately 10.8M ops/s, followed by WriteBoost RCU at 8.4M ops/s, std::shared_mutex at 7.0M ops/s, and Folly RCU at 4.6M ops/s. This demonstrates that WriteBoost RCU's optimizations help minimize the overhead of RCU's copy-on-write approach for small data structures.

As the protected data size increases, several distinct patterns emerge:

1. Folly RCU shows the most dramatic performance degradation, with throughput dropping by three orders of magnitude from 4.6M ops/s at 2 integers to just 10.7K ops/s at 65,536 integers. This sharp decline reflects the increasing cost of memory allocation and copying in traditional RCU implementations.

2. WriteBoost RCU maintains competitive performance

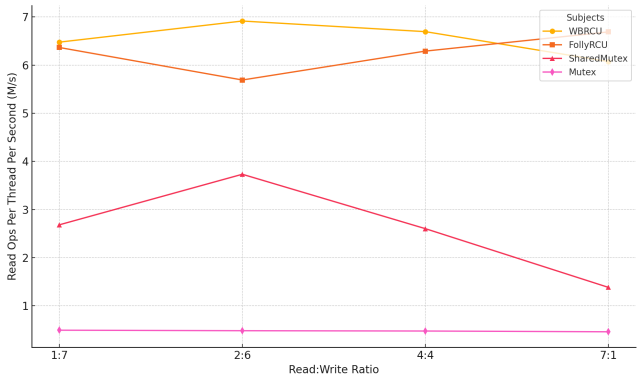

Figure 5: Read Operation Per Thread with Increasing Read-write Ratio

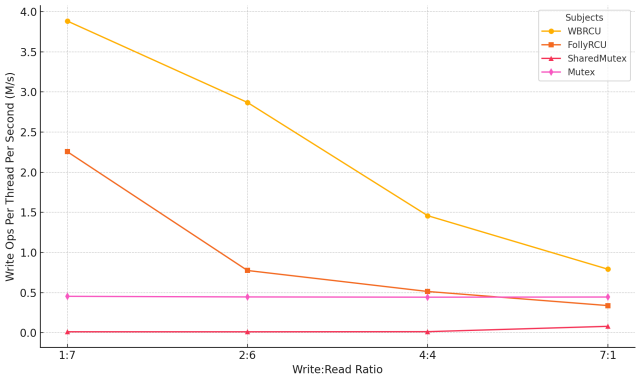

Figure 6: Write Operation Per Thread with Decreasing Read-write Ratio

up to moderate data sizes (around 4,096 integers), achieving 6.5M ops/s. The object pool and batch update optimizations effectively amortize the copying and memory management overhead. However, performance does begin to degrade with very large data sizes, dropping to 762K ops/s at 65,536 integers.

3. Both mutex-based approaches show remarkable stability across all data sizes. std::mutex maintains around 8-10M ops/s, while std::shared_mutex consistently delivers 5-7M ops/s. This stability is expected as these mechanisms don't require data copying during updates.

The benchmark illustrates that data structure size has a significant impact on RCU write performance, with larger sizes increasing the overhead of copy operations and memory management. While traditional RCU implementations like Folly RCU show rapid performance degradation as size increases, WriteBoost RCU's innovations - particularly its object pool and batch processing - help mitigate this cost, allowing it to maintain competitive write performance for data structures up to a few kilobytes in size.This extends the practical utility of RCU beyond just tiny data structures, though the fundamental limitations of copy-on-write do eventually become apparent with very large data structures.

These results suggest that when considering RCU for a particular use case, the size of the protected data structure should be a key factor in the decision-making process, potentially even more important than other traditional considerations like read-write ratios. WriteBoost RCU's optimizations help push these size limitations further, but they cannot completely eliminate the inherent scaling challenges of copy-on-write operations.

4.4 Varying Read-Write Ratio

To evaluate how different synchronization mechanisms handle mixed workloads, we measured both read and write performance across varying ratios of readers to writers, maintaining a total of 8 threads. Each operation includes 100ns of simulated work to represent realistic workload characteristics. Figures 5 and 6 illustrate the throughput for readers and writers respectively under these varying ratios.

The read performance results (Figure 5) show that both RCU implementations maintain consistently high throughput across all ratios, with WriteBoost RCU achieving around 6-7M ops/thread/second and Folly RCU performing similarly. This stability is particularly noteworthy as it demonstrates that RCU readers maintain their performance regardless of write contention. In contrast, std::shared_mutex shows declining performance as the proportion of writers increases, dropping from 3.7M to 1.4M ops/thread/second, while std::mutex maintains consistently poor read throughput around 480K ops/thread/second due to its strict mutual exclusion.

The write performance results (Figure 6) reveal several surprising insights. WriteBoost RCU achieves the highest write throughput, starting at 3.9M ops/thread/second with a single writer and gradually declining to 792K ops/thread/second with seven writers. Folly RCU, while showing lower absolute performance, still maintains better throughput than std::mutex when the number of readers and writers are equal (4:4 ratio). This is particularly noteworthy as Folly RCU lacks any specific update-side optimizations. However, in write-heavy scenarios (7:1 write:read ratio), Folly RCU's performance does fall below std::mutex, as the overhead from memory allocation, copying, and reclamation begins to outweigh the benefit of non-blocking writes.

The poor write performance of std::shared_mutex, while notable, is expected behavior. In read-heavy workloads, writers suffer from starvation as they must wait for all concurrent readers to complete before proceeding. This is a well-known limitation of reader-writer locks that RCU's design specifically addresses by allowing writers to proceed without blocking readers.

These results challenge the traditional view that RCU im-

plementations necessarily sacrifice write performance for read performance. Even without specific write-side optimizations, RCU can provide competitive write throughput in balanced workloads, though this advantage diminishes in write-heavy scenarios or when working with larger data structures where copy and memory management overhead becomes more significant. WriteBoost RCU's optimizations help extend this performance advantage across a broader range of workload patterns.

5 Conclusion

This paper presents WriteBoost RCU, a novel RCU library implementation that challenges traditional assumptions about RCU's applicability while improving both performance and programmability. Our research leads to several important conclusions about RCU's potential and future directions.

First, our evaluation reveals that RCU's performance benefits extend beyond the commonly assumed read-heavy scenarios. While RCU is often recommended primarily for workloads where read performance is crucial, our benchmarks demonstrate that RCU can provide competitive write performance even in balanced workloads. Even without specific write-side optimizations, Folly's RCU implementation achieved higher writer throughput than mutex-based synchronization with equal numbers of readers and writers. This suggests that RCU's fundamental design - where neither readers nor writers block each other - can benefit both operation types. These findings indicate that RCU's potential use cases may be broader than traditionally thought, warranting further exploration of opportunities to optimize update-side performance.

Second, our research demonstrates that incorporating update synchronization responsibility into the RCU library itself yields significant benefits without compromising RCU's exceptional read performance. By managing all aspects of synchronization within the library, WriteBoost RCU not only improves update and reclamation efficiency but also substantially enhances programmability. This comprehensive approach eliminates the need for users to implement complex synchronization strategies manually while maintaining RCU's core performance characteristics.

Finally, the effectiveness of WriteBoost RCU's RCU-centric update-side synchronization mechanism is evident in our performance evaluations. The combination of a single updater/reclaimer design, batch updates processing, and an object pool for memory management significantly improves update-side performance compared to traditional RCU implementations. This demonstrates that RCU libraries can successfully optimize write operations while preserving their fundamental benefits.

These findings suggest that the future of RCU lies not just in its traditional role as a specialized tool for read-heavy workloads, but as a more versatile synchronization mechanism capable of delivering high performance across a broader range of scenarios. The success of WriteBoost RCU's approach indicates that there may be additional opportunities to enhance RCU implementations by rethinking traditional design boundaries and responsibilities.

References

[1] DESNOYERS, M., MCKENNEY, P. E., STERN, A. S., DAGENAIS, M. R., AND WALPOLE, J. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems 23*, 2 (2012), 375–382.

[2] FACEBOOK OPEN SOURCE. Folly: Facebook open-source library, 2023.

[3] MATVEEV, A., SHAVIT, N., FELBER, P., AND MARLIER, P. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 168–183.

[4] MCKENNEY, P. E., ET AL. Read-copy update (rcu). Tech. Rep. P2545R4, ISO/IEC JTC1/SC22/WG21, March 2023.

[5] SIAKAVARAS, D., NIKAS, K., GOUMAS, G., AND KOZIRIS, N. Rcu-htm: Combining rcu with htm to implement highly efficient concurrent binary search trees. In *2016 IEEE International Conference on Parallel, Distributed, and Network-Based Processing (PDP)* (2016), pp. 127–134.

[6] WIKIPEDIA. Read-copy-update, 2023.