# Evaluating the Read-Copy Update Fine Grained Synchronization Technique on Modern Systems

Gustavo Silvera
15-740 Project, Fall 2022

*Introduction:*

Read-Copy Update (RCU) is a synchronization mechanism for parallel programs that minimizes the performance impact of concurrent reads while guaranteeing memory coherence. Unlike conventional locking primitives that provide mutual exclusion between concurrent threads, RCU allows reads and writes to be performed in parallel, and the reads have nearly zero synchronization overhead [1]. This final project implements various benchmarks with different data structures and thread allocations to compare the performance of RCU with other synchronization mechanisms such as traditional locks and atomics.

*Background & Motivation:*

Synchronization is a crucial part of parallel programming to avoid correctness issues with race conditions. A common disadvantage of many synchronization flavors is a large performance impact due to enforcing mutual exclusion forces serialization of some critical sections of the program. Therefore the size of this critical section increases the Amdahl bottleneck and limits speedup gains from parallelism. It is therefore desirable to minimize the size and impact of these critical sections to avoid this scalability bottleneck. However, the difficulty in optimizing parallel programs this way creates a tradeoff between performance impact and its ease of implementation/integration into a program.

The two main traditional techniques this project explores beyond RCU is locking and atomics. Locking is a simple synchronization method that enforces mutual exclusion where only one thread has exclusive access to some data because of barriers at the start and end of a critical section. Atomic instructions are typically implemented as additional instructions in the instruction set architecture (ISA) that operate on individual primitives such as ints, floats, longs, etc. and enable all operations to be performed coherently (i.e. all at once, no in-betweens). While atomics are generally much faster than locking, they are also more restrictive as they can only operate on individual fixed-length data types.

In October of 2002, the RCU synchronization mechanism was introduced to the Linux kernel. It promises scalability improvements by enabling near-zero overhead reads and concurrent writes. By tracking existing references to some shared data, the RCU backend knows which threads need access to the old data so writers can exclusively publish a modified copy of the data that new references see, and readers need no additional synchronization. Only existing references will continue to see the unmodified "old" data and new references will see the newly modified data. Once all existing references have completed their work, the writer can safely reclaim the memory from the old data it has replaced. Since writes are significantly more expensive now, as they need to copy the memory of the global data, modify it, and wait for existing readers to finish before reclaiming memory, RCU thrives in write-infrequent situations where read latency is paramount. RCU has since been used often throughout the Linux kernel, notably in the networking and drivers subsystems which often encounter read-mostly situations.

*Design:*

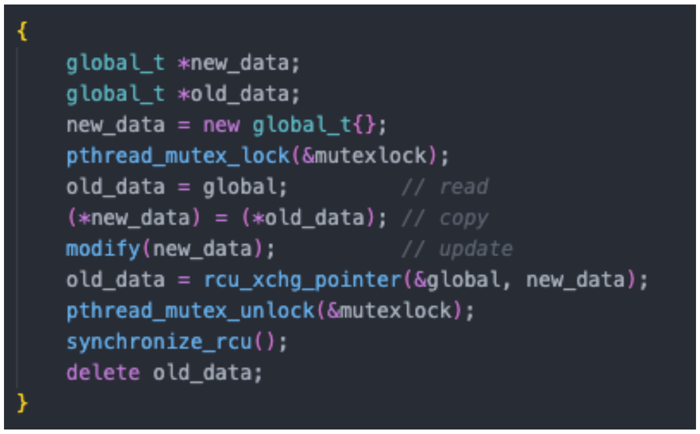
This final project implements various benchmarks with readers and writers operating concurrently on some data, with parameterized thread distributions. The writer threads run alongside the readers for the entire duration of the program with delays between consecutive writes to follow the "read-mostly write infrequently" situation that RCU prefers. This final project explores five synchronization techniques including: RCU, reader-writer locks, mutex locks, atomics, and none (race condition). Note that the "no synchronization" race condition fails to provide correctness guarantees, but is intended to demonstrate the best-case performance scenario. Additionally, the write operations depend on the shared data structure which this project explores: bumping an int (single counter), bumping three ints (struct containing three ints), updating a string, and modifying a dynamically sized array (vector).

*Implementation:*

This project uses the open source realtime userspace RCU library [3] to implement the fine-grained locking of RCU in the benchmarks. Programs were written in C++17 which includes support for the `std::atomic<T>` [4] primitive, `pthread_rwlock_t` for reader-writer locks, and `pthread_mutex_t` for mutual exclusion locks.

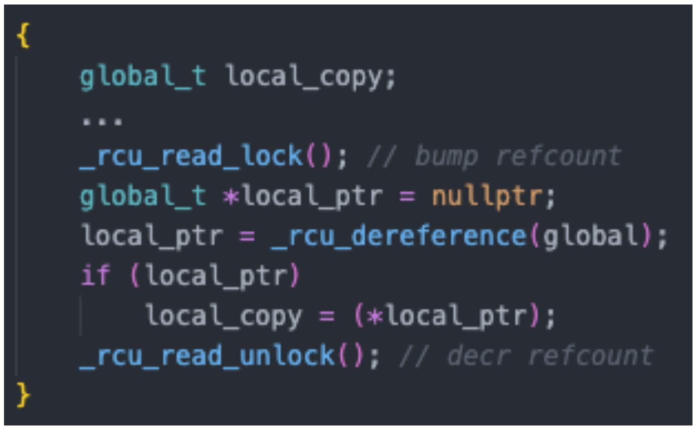
The scaffold code for my benchmarks is contained in the primary source file benchmark.cpp which implements the primary reader and writer threads and all the RCU necessary initialization and boilerplate code. The interface implementations for each data structure operation is held in separate header files in operations/*.h to conceptually separate the various operations within separate binaries.

All the operation headers implement the same interface for write_op, read_op, and finalize_op, and they are somewhat similar, especially in the RCU case. Writes using RCU are performed similarly to Figure 1a, where a mutex allows the writer to read and copy the global data, make local changes, then atomically exchange the pointer and synchronize (wait for readers) until freeing the old data. Reads are performed as in Figure 1b where the `_rcu_read_lock()` and `_rcu_read_unlock()` add very minimal overhead by incrementing and decrementing the refcount of this global so writers can keep track of it.



```
{
    global_t *new_data;
    global_t *old_data;
    new_data = new global_t{};
    pthread_mutex_lock(&mutexlock);
    old_data = global;          // read
    (*new_data) = (*old_data); // copy
    modify(new_data);           // update
    old_data = rcu_xchg_pointer(&global, new_data);
    pthread_mutex_unlock(&mutexlock);
    synchronize_rcu();
    delete old_data;
}
```

Figure 1a: Typical RCU write scaffold code



```
{
    global_t local_copy;
    ...
    _rcu_read_lock(); // bump refcount
    global_t *local_ptr = nullptr;
    local_ptr = _rcu_dereference(global);
    if (local_ptr)
        local_copy = (*local_ptr);
    _rcu_read_unlock(); // decr refcount
}
```

Figure 1b: Typical RCU read scaffold code

Of the various data structure operations implemented for this project, the simplest are for the bump-counter (bump_counter.h) which is implemented as a global `uint64_t` where the write operations increment the value by one. This can be done efficiently with an atomic [4] which fully utilizes the performance advantages of an atomic instruction for loads and stores. Next, an implementation with a struct containing three integers (struct_abc.h) is used to highlight a simple case where individual atomics cannot provide the same guarantees since the struct's update function bumps all three components at once and atomics can't guarantee execution order/coherence. An std::string is used to implement the "atomic string" operation (atomic_string.h) where the update function overwrites the string entirely with the current time. Finally an `std::vector<int>` (atomic_vector.h) is used as a complex data structure where the write process includes bumping a random index and extending the length of the vector by 1 with 10% probability, showcasing a complex data structure that can both resize and update in-place.

*Evaluation Methodology:*

The performance evaluation for these various parametrized benchmarks was done by writing a python script to execute each of the executables with all combinations of parameters for thread distributions (from 1 to 30 reader/writer threads), synchronization modes, and operations. The data for average number of cycles per reads and writes was accumulated and presented. This data collection was performed on my personal machine running an Apple arm64 M1 MAX 10 core heterogenous (8 big + 2 small) CPU with 64GB unified memory, this data was collected on the macOS Monterey 12.6 OS.

*Results:*

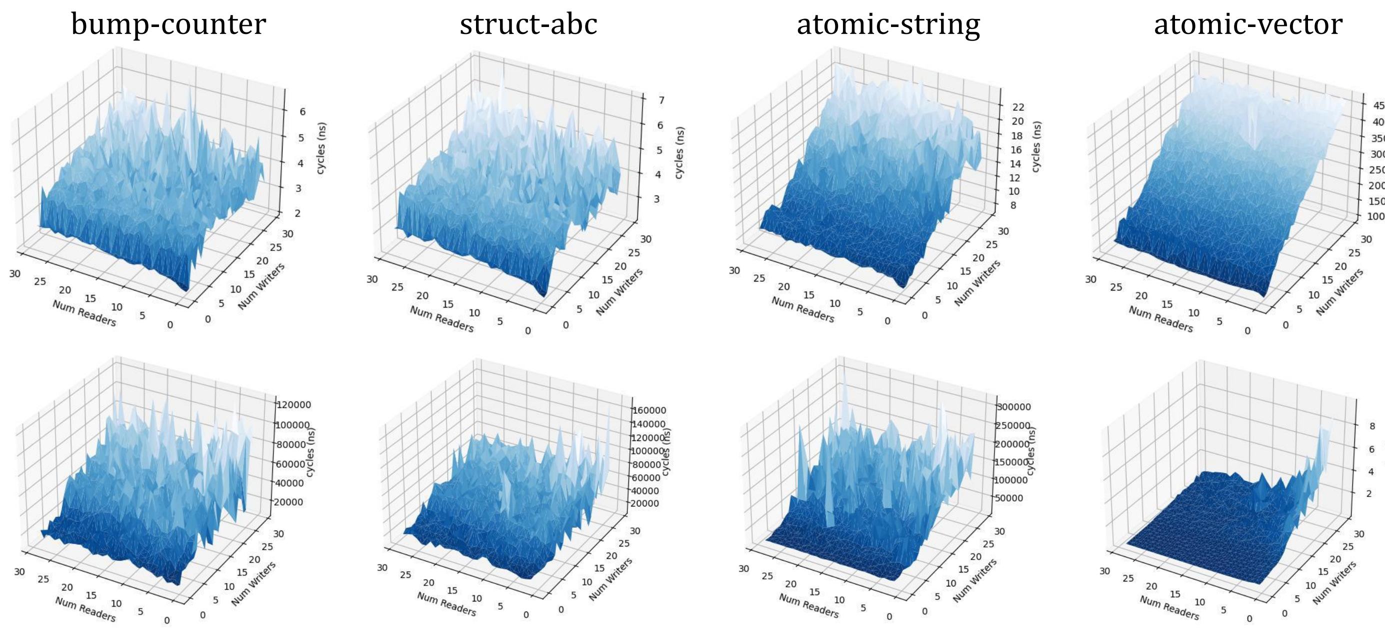


Figure 2: RCU latency mountain graphs (lower is better) for reads (top row) and writes (bottom row). X-axes indicate the number of readers and Y-axes indicate the number of writers.

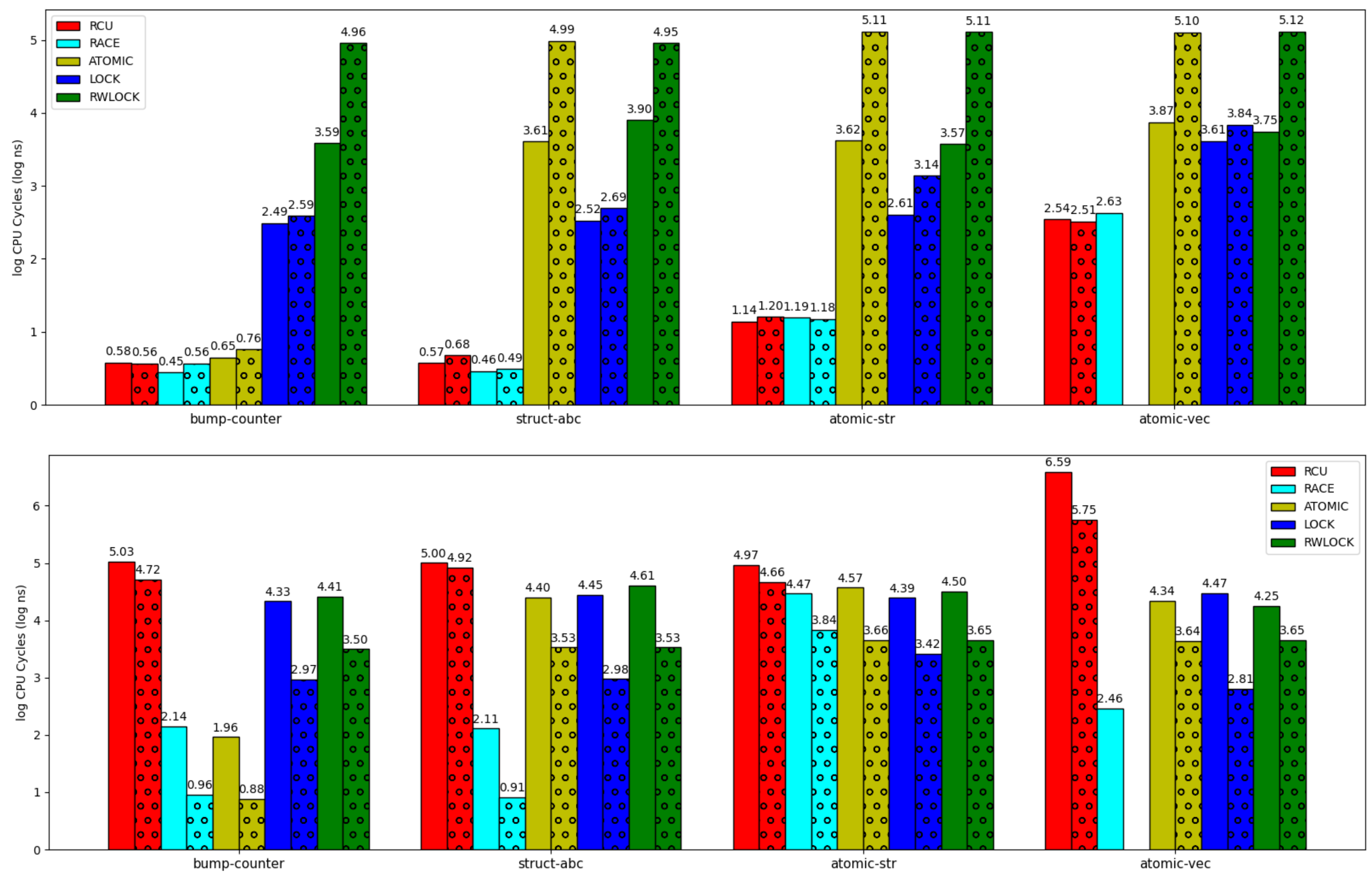


Figure 3: Relative (log10) latency performance graphs (lower is better) for synchronization methods grouped by operation. Showcasing read (top) and write (bottom) latencies. Plain bars indicate a 20 reader & 2 writer setup, dotted bars indicate a 20 readers & 30 writers setup (worse case). Atomics (yellow) for non bump-counter ops were implemented as reader-writer locks.

*Discussion:*

RCU performed as advertised and achieved impressive results when the situation was favorable. As seen in Figure 2, the read latencies of RCU generally did not scale with the number of reader threads indicating strong scalability results for read-oriented programs. However, read latencies did scale linearly with the number of writer threads (more on complex data structures with expensive update operations) which follows the expected behavior. The write latencies from Figure 2 demonstrate significantly higher cycles per operation (>= 20000 cycles) compared to the handful of cycles for all read operations. Especially for complex operations (atomic-string/vector), the write latencies scale with contention and operation complexity. Since RCU needs to perform memory operations in a critical section (for exclusive write access) of large pieces of data, it is reasonable that there is a larger performance penalty with the larger critical sections.

*Surprises:*

While not surprising that RCU had the worst read performance across the board (Figure 3), sometimes orders of magnitude higher latencies than the competition, it was interesting to see write performance remain unchanged in the situation of increased write contention (20r + 30w). I believe this is because there are sufficiently many readers in this situation where the write latency is bottlenecked by waiting for existing readers' synchronization. Additionally, the write latencies for all synchronization schemes in the second scenario (20r + 30w) had slightly lower latencies compared to when there were fewer writers (20r + 2w). This makes me wonder if there is some write buffering optimization happening with high contention as the per-write overhead is lower.

It was also surprising that RCU was able to beat the performance of the race condition for some cases with complex reads, showcasing the true "zero-overhead" promise of RCU reads from the start.

Additionally, I'm not sure why the reader-writer locks seemed to perform significantly worse than the mutual-exclusion mutex lock. Reader-writer locks simultaneous read accesses or an exclusive write access, while mutex locks allow only exclusive access regardless of access type. I would have hypothesized that the reader-writer locks would perform better in read-mostly situations.

Furthermore, the empty cyan bar for the race-condition measurement of atomic-vector in Figure 3 was due to a program crash, likely because of the race condition that occurs on frequent writes which may involve resizing the vector.

*Insights:*

What these results indicate is that RCU can be a perfectly viable and desirable alternative to other concurrency control mechanisms when the problem description falls under the category of read-mostly parallelism. While read performance is similar to atomics and write performance is much worse as seen in Figure 3, RCU provides significantly more flexibility than atomics which could only be used in the simplest benchmark of bumping a single integer. In cases where shared data structures are more complicated than individual data types, memory coherence between threads is paramount, minimizing read latency is desired, write latency is either unimportant or highly infrequent, and the write operations are not too memory intensive, RCU becomes the best option to satisfy these goals.

*Proposal Goal Satisfaction*

According to the initial proposal for this final project, the 100% goal was to use and compare user-space RCU against other concurrency algorithms with various reader/writer thread distributions. This goal was achieved as I provided implementations for several concurrency control algorithms (mutexes, atomics, reader-writer locks, none, & RCU) across various operations that were all evaluated and compared. There was an additional idea in the proposal to compare the RCU performance on various operating systems: macOS vs Linux, but I found no significant difference as the library operates in user space with little OS interaction. The 125% goal was to compare the performance of RCU on various architectures, such as a modern x86-64 versus arm64 as described here [5] but I did not manage to find enough time to accomplish this.

## References

[1] McKenney, P. 2007. What is RCU, fundamentally? [LWN.net]. https://lwn.net/Articles/262464/.

[2] What is RCU? – "Read, copy, Update". What is RCU? – "Read, Copy, Update" - The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html.

[3] Userspace RCU. https://liburcu.org/. (https://github.com/urcu/userspace-rcu)

[4] Std::Atomic. cppreference.com. https://en.cppreference.com/w/cpp/atomic/atomic.

[5] Desnoyers, M. 2019. The 5-year journey to bring restartable sequences to linux. EfficiOS. https://www.efficios.com/blog/2019/02/08/linux-restartable-sequences/.