Read-Copy Update in a Garbage Collected Environment

Harshal Sheth, Nihar Sheth, Aashish Welling

March 1, 2017

Abstract

Read-copy update (RCU) is a synchronization mechanism that allows efficient parallelism when there are a high number of readers compared to writers. The primary use of RCU is in Linux, a highly popular operating system kernel. The Linux kernel is written in C, a language that is not garbage collected, and yet the functionality that RCU provides is effectively that of a "poor man's garbage collector" [1]. RCU in C is also complicated to use, and this can lead to bugs. The purpose of this paper is to investigate whether RCU implemented in a garbage collected language (Go) is easier to use while delivering comparable performance to RCU in C. This is tested through the implementation and benchmarking of 4 linked lists, 2 using RCU and 2 using mutexes. One RCU linked list and one mutex linked list are implemented in each language. This paper finds that RCU in a garbage collected language is indeed significantly easier to use, has similar overall performance to, and on very high read loads, outperforms, RCU in C.

1 Introduction

For much of computing's history, it was a given that clock speeds would increase exponentially over time. From the mid-1970s to the early 2000s, they did increase at that rate, as shown in fig. 1. Programmers could therefore count on their programs running faster and faster, year by year, purely from clock speedups. Unfortunately, exponential increases in clock speeds are now a thing of the past. Today, the new trend in hardware is increasing the number of cores available to programmers [2, pp. 11-12]. To harness the power of multi-core computing, however, smart parallelized design is essential. This is difficult for a host of reasons, not the least of which is related to shared memory access, as we explain next.

1.1 Unprotected Data Access

A major difficulty when programming with multi-core architectures is that it is not safe for multiple threads to access shared memory simultaneously. If multiple threads try to write to the same memory at the same time, or some try to read while others try to write, the data can be corrupted. For example, if one thread is updating the value of a linked list element, and another thread tries to read the value of that element while the first thread is partway finished updating, the reader thread will get a partially updated bogus value (neither the old nor the new). This situation is known as a race condition and is diagrammed in fig. 2 [4]. Synchronizing parallel threads to prevent these types of situations from occurring is crucial, and there are several ways to do it.

1.2 Atomic Operations

Atomic operations are a synchronization mechanism that can be used for very basic operations, such as loading and storing values, and doing simple addition and subtraction. If an operation is

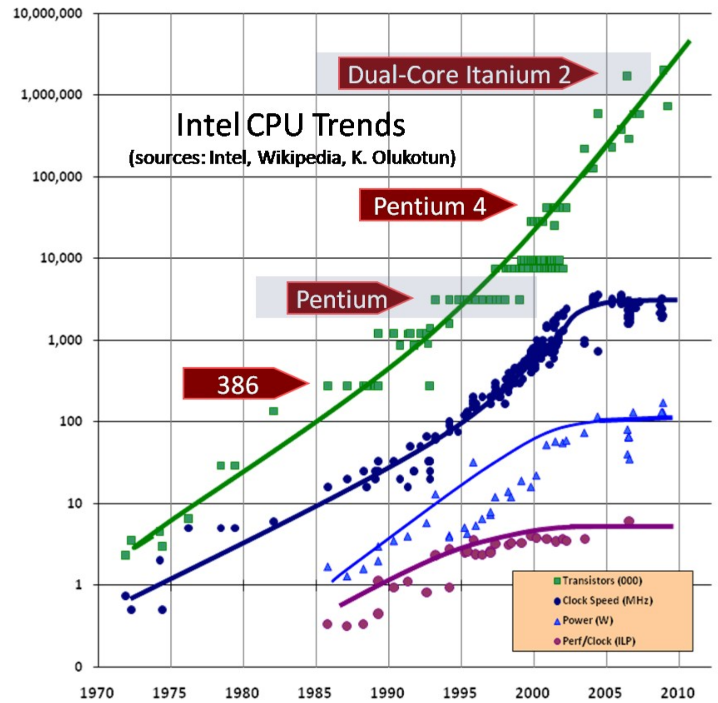

Figure 1: The dark blue line shows clock speeds of Intel CPUs over time. Until the early 2000s, speeds increased exponentially but have now begun to flatten out.

Source: [3]

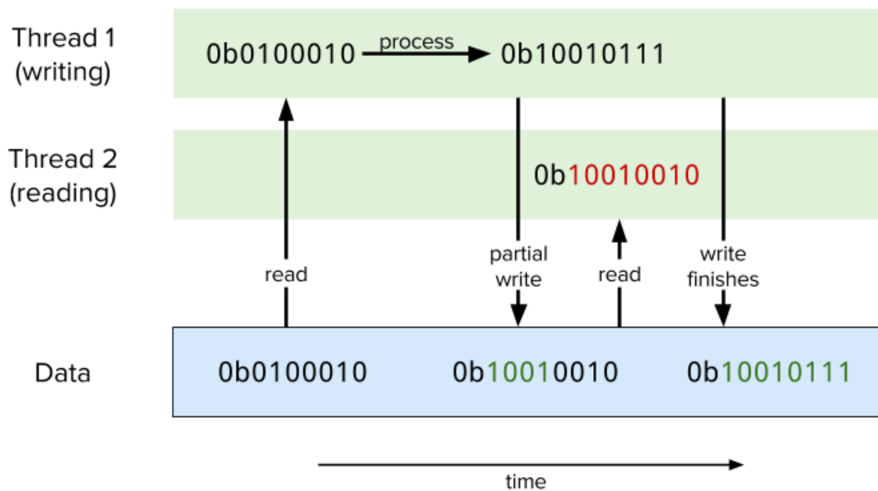

Figure 2: Thread 1 is in the process of writing the value of a linked list element when Thread 2 reads. Thread 2 ends up reading the partially written bogus value (`0b10010010`) instead of reading either the original value (`0b0100010`) or the new value (`0b10010111`).

atomic, it is guaranteed that the operation will not be interrupted. This means that if two threads are operating atomically on some common data, then the operations will not conflict, since neither operation can interrupt the other. Thus, if all access to some memory is done atomically, there is no risk of corrupting that data. For example, writer Thread 1 in fig. 2 would not be interrupted by reader Thread 2, removing the possibility of a race condition.

However, atomic operations are limited to primitive data types and pointers. In addition, multiple atomic operations are not extremely useful, as other operations could be interleaved between them, defeating their purpose. Therefore, they are not useful in conjunction with more complicated data structures, especially those that have multiple data fields or need to do multiple operations during updates [4]. For these larger and more complicated use cases, another synchronization mechanism must be used.

1.3 Locking

Locking is a synchronization mechanism that prevents multiple threads from using certain memory at the same time. It is the most commonly used synchronization mechanism. Locking is commonly achieved using mutexes. However, mutexes do not perform optimally in some use cases because they require exclusivity. If multiple threads attempt to access some shared data concurrently, then the mutex will force these threads to access the data one at a time, effectively serializing these threads [5]. This serialization of threads is detrimental to performance in any multithreaded system. Even read-write mutexes (RW mutexes), which allow multiple threads to read the same data in the absence of a writing thread, are suboptimal. The presence of a single writing thread prevents all other threads from reading or writing to the same memory. For the many modern applications that have far more readers than writers, the ideal solution allows readers to operate continuously, regardless of the presence of writers.

1.4 Read-Copy Update

Read-copy update, or RCU, is an algorithm used to achieve this lock-free reading at the cost of more expensive writes. In applications with numerous readers and relatively few writers (such as caches), the additional writer overhead is insignificant compared to the performance improvements from lock-free reads.

1.4.1 A Basic Example

RCU works by copying data, updating that copy, and atomically switching the old data with the updated copy. Figures 3 to 5 illustrate how Element B in the sample linked list would be updated with RCU. Each figure shows a step of the RCU algorithm. The red lines indicate that a writer thread could be on the element, while the green lines indicate a reader thread could be present.

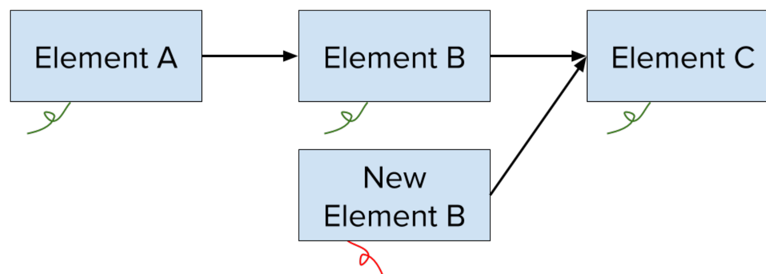

Figure 3: RCU Step 1

In fig. 3, element B is copied and updated with the new value, and a pointer is created from the new element to C.

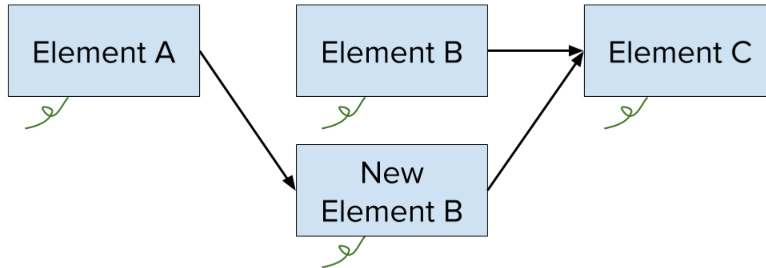

Figure 4: RCU Step 2

In fig. 4, the pointer from A to B is atomically switched to point from A to new element B. At this point, there could be readers on any of the four elements.

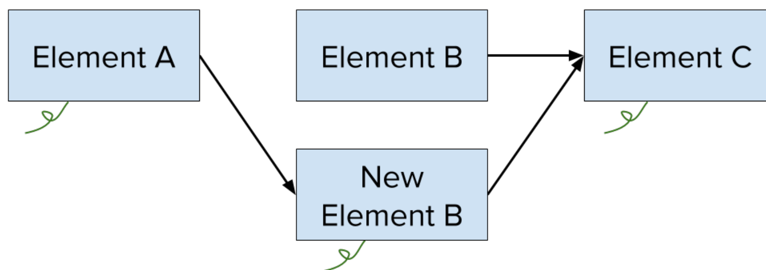

Figure 5: RCU Step 3

After all existing readers on the old element B have moved on, there will be no readers left on it since there is no pointer to old element B. At this point, it is safe to reclaim element B's memory. This is shown in fig. 5.

1.4.2 Quiescent States and Grace Periods

How does RCU detect when all readers have left the old element (when it is safe to free from memory)? It uses quiescent states and grace periods. A quiescent state is a time period during which a particular thread is not reading the shared data structure. A grace period is a time period during which all threads have gone through at least one quiescent state. After a grace period, it is safe to free old versions of elements, because all reader threads that could have been on the old element would have finished reading, and any new reader threads would go exclusively to the updated copy.

1.4.3 Writers in RCU

One disadvantage to RCU is that the writing has a significantly higher cost than it does with RW mutexes. Writers are not only serialized like they are with mutexes but also they must copy the data before even beginning to modify it. They also must use atomic operations, which do have some small added cost. Thus, RCU is only used when there are relatively few writers compared to readers.

1.4.4 Current RCU Implementations

The most notable user of RCU is the Linux kernel, where fig. 6 shows that its usage has been growing since its introduction in 2002. The Linux kernel, like most other implementations and

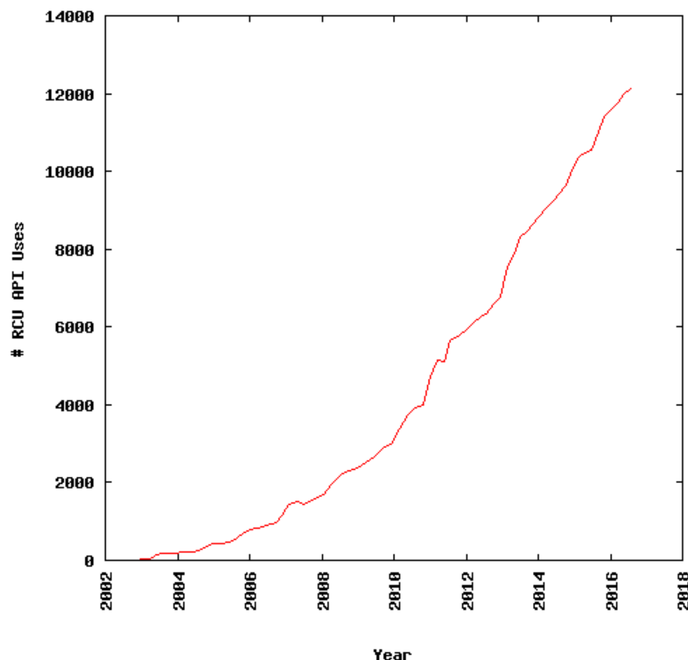

Figure 6: RCU usage in the Linux kernel has been growing exponentially over time. Source: [8]

users of RCU, is written in C [6, 7], a language that is not garbage collected.

1.5 RCU in a Garbage Collected Language

While part of the RCU algorithm deals with copying data and atomically switching pointers, a large part of it deals with determining when it is safe to free memory (using quiescent states). In garbage collected languages, freeing unused memory is the role of the garbage collector. For a large part, RCU plays the role of a garbage collector. In fact, Paul E. McKenney, the inventor of RCU, calls it “a poor man’s garbage collector” [1].

RCU in C is complicated to use. In fact, RCU users in Linux must comply with a 17-point checklist of conditions under which RCU is safe to use, and “violating any of [those] rules [...] will result in the same sorts of problems that leaving out a locking primitive would cause [data corruption]” [9]. These rules primarily to tracking memory usage for the purposes of reclaiming at the right time. The usage complexity has to led a large number of bugs [10, 11], even in the well-maintained Linux kernel.

If RCU were implemented in a garbage collected language, there would be no need to keep track of quiescent states and grace periods, since the garbage collector would handle freeing memory. This would greatly simplify RCU code, since much of the “poor man’s garbage collector” could be replaced by the language’s garbage collector. In addition, users of RCU would not need to comply with as strict guidelines for usage, since the garbage collector would track unused memory. Depending on the efficiency of the garbage collector, this approach could even have comparable performance to RCU in C.

1.6 Purpose

To address the shortfalls of read-write mutexes, as well as the problems with RCU in C, this paper investigates whether or not RCU in a garbage-collected language is a viable option. We will consider it viable if it is simpler and easier to use than its C counterpart, while still providing similar performance benefits to C RCU.

1.7 Programming Language Selection

We chose Go as the garbage collected language to use for the purpose of this paper. There are two reasons why we chose Go. The first is that, out of all garbage collected languages, Go is the most usable for systems level programming since it allows access to certain low-level functions, such as pointer arithmetic, that many other garbage collected languages do not. This functionality is necessary to implement RCU. Secondly, Go's garbage collector is very advanced, built "not only for 2015 but for 2025 and beyond" [12]. Its garbage collector is specifically built for multi-threaded applications.

2 Methods

We used two metrics for evaluation: the ease of use of RCU in Go compared to that of RCU in C and the performance of RCU in Go as compared to that of RCU in C. To do this, we used four thread safe singly-linked lists, 2 in Go and 2 in C. In each language, one linked list used RCU for synchronization and the other used mutexes. We chose to use singly-linked lists for the purposes of this paper as they are the most commonly used data structure with RCU [9]. We evaluated ease of use by analyzing the difficulty of writing the code utilizing C RCU and Go RCU. The implementation of the RCU library is not relevant since it only needs to be written once; RCU users are not concerned with it. As shown in fig. 7, we compared the performance of C RCU directly with that of Go RCU, as well as the performance improvement over mutex given by introducing RCU in each language.

2.1 RCU in User Space

Because we could not find any fully functional POSIX-compliant kernels written in Go, it was not possible to test Go RCU in kernel space. Therefore, we experimented on RCU in user space. User space applications have access to a limited section of memory, can be interrupted[1], and cannot perform all operations that kernel space code can. However, for this experimentation, these limitations were not of issue.

2.2 C Implementation

There were two C linked lists: one using a read-write mutex and the other using RCU[2].

RW Mutex The read-write mutex implementation was fairly simple. We used a read-write mutex from the POSIX threads (Pthread) multithreading library to protect a standard linked list.

[1] Kernel space applications can also be interrupted on preemptive kernels. Many modern kernels, including recent versions of Linux, are preemptive kernels [13].

[2] The benchmarking code was written in C++, but the linked lists and RCU implementation were in C.

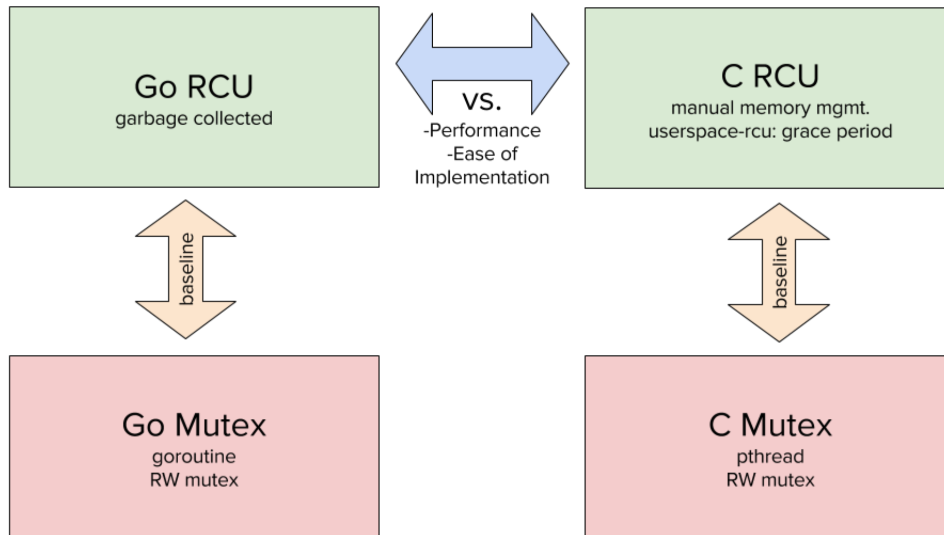

Figure 7: The Go and C RCU linked lists were tested directly against each other. They were each also tested against their respective mutex linked lists to factor out programming language performance.

RCU The RCU linked list was considerably more complicated. Since both implementations were to be tested in user space, we did not use the Linux kernel RCU implementation. Because of the difficulty of implementing and using RCU in C, we used an RCU library called *userspace-rcu* [14]. Desnoyers has shown that the user space RCU implementations in this library have good performance [7]. This library, which is written in C, follows the standard RCU API by providing the functions outlined in table 1. We used this API to implement the linked list in C.

2.3 Go Implementation

Two additional listed lists were implemented in Go, one using an RW mutex and other using RCU.

RCU Library The first step in creating these linked list implementations was to write an RCU library in Go (a Go equivalent of *userspace-rcu*). Initially, the Go library followed the standard RCU API (see table 1), but we realized that a strict adherence to the said API would not be logical; many of the functions provided by that API were not needed because of the Go garbage collector. The `rcu_read_lock()` and `rcu_read_unlock()` methods relate to the detection of quiescent states, and the `synchronize_rcu()` and `call_rcu()` methods relate to the handling of grace periods and memory cleanup. Go's garbage collector eliminated the need for tracking quiescent states and manually freeing memory. Thus, Go's RCU API simply consisted of `rcu_assign_pointer()` and `rcu_dereference()`, each of which were implemented with a single atomic operation.

RW Mutex We wrote a standard linked list using the RW mutex from Go's `sync` package. Multithreading was done with Go's *goroutines.*

RCU The Go RCU linked list was similar to a standard linked list implementation, except that it used `rcu_dereference()` when reading a pointer to an element in the linked list, and used `rcu_assign_pointer()` when updating a pointer's value. It did use a simple mutex to serialize

Table 1: Core RCU API
Source: [9]

API Function	Purpose
`rcu_read_lock()`	Mark beginning of read-side critical section
`rcu_read_unlock()`	Mark end of read-side critical section
`synchronize_rcu()`	Block until all existing read-side critical section on all CPUs are completed
`call_rcu()`	Callback version of `synchronize_rcu()`; call specific function when all existing read-side critical section complete
`rcu_assign_pointer()`	Safely store RCU-protected pointer so that there is no memory reordering around it
`rcu_dereference()`	Safely load pointer without reordering (does not dereference the pointer; simply s it so it is ready for dereferencing)

the writers, as is required by RCU. Like the RW mutex implementation, it used *goroutines* for multithreading.

2.4 Benchmark Setup

2.4.1 Benchmarking Machine

All performance testing was conducted on an Ubuntu 14.04 machine with Linux 3.13.0, 16GB of memory, and an Intel Xeon Quad-Core Dual Socket processor. The machine had 8 logical cores.

2.4.2 Benchmarking Procedure

The application used for benchmarking the four linked lists works as follows:

1. Linked list of size 500 is initialized with random filler values.
2. 7 new work threads are initialized (goroutines in Go, Pthreads in C++).
3. 8 worker threads (7 child, plus main thread - one thread per core) each concurrently perform 10 million iterations of their loop. In each iteration, each thread has a probability given by its mix of updating a random node on the common linked list. If it doesn't update, it will simply read a random value. A higher mix means there will be more updates.
4. Steps 2-3 are timed.
5. We ran the benchmark application (steps 1-3) with different mixes of updates vs. lookups: 0%, 1%, 2%, 3%, 5%, 7%, 10%, 15%, 20%, and 30%. In practice, RCU would not be used for mixes of greater than 10% [15], but we benchmarked mixes up to 30% to gain a more complete understanding of the data.

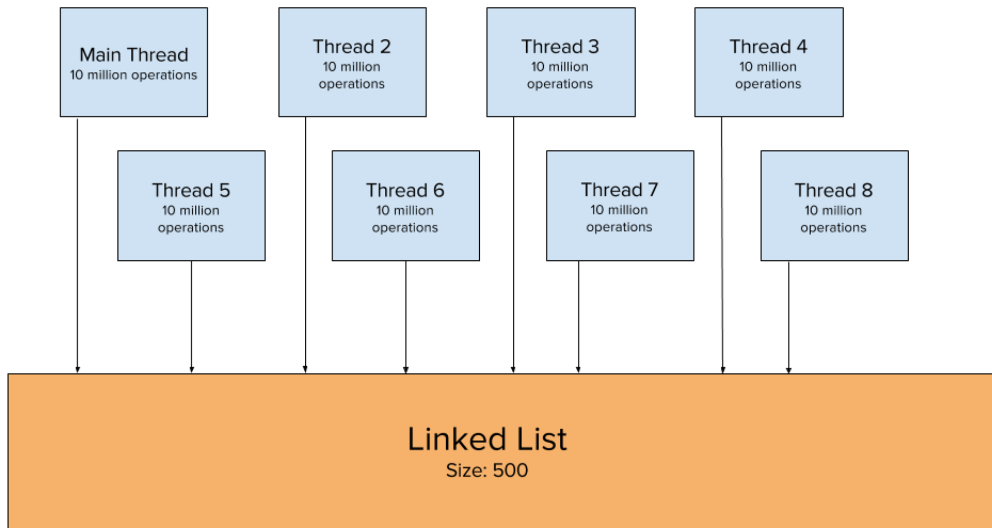

Figure 8: For each mix, each of the 8 threads did 10 million read/write operations on a linked list of size 500. The fraction of each thread's operations that were reads or writes was determined by the mix.

3 Results and Discussion

3.1 Performance

3.1.1 Go RCU vs. C RCU

The first experiment which was performed was the test of the Go RCU linked list directly against the C RCU linked list, the results of which are shown in fig. 9. Despite Go being a higher-level language, Go RCU actually performed faster than C for mixes of less than approximately 3%. For a 3% mix, Go RCU completed the operations in 19.468 seconds, compared to C RCU, which completed in 21.1 seconds. After 3%, Go RCU's run time appears to increase linearly, while C RCU's flattens out for some time before increasing linearly. At the point of maximum run time difference (a mix of 30%), Go RCU runs in 145.733 seconds, compared to C RCU which runs in 87.9 seconds. However, RCU would not be used for this high of a mix; in fact, RCU is normally only used for mixes of 10% or less [15], for which Go RCU performs comparably.

The likely reason for Go's outperformance for very low mixes is that Go's garbage collector has a smaller overhead than C RCU. C RCU must set up additional structures to detect quiescent states and grace periods, and call cleanup callbacks accordingly. In contrast, Go's garbage collector is built into the language itself, so it has relatively less overhead for high read loads. Thus, the performance of Go was superior to that of C for low mixes. However, for larger mixes, the overhead of RCU in C would be negated by the superior performance of C as a language, and thus C RCU begins to perform better than Go RCU.

3.1.2 RCU vs. Mutex Comparisons

In the second experiment, the performance of the Go RCU linked list was compared against the performance of the Go read-write mutex linked list. The same comparison was done for C. This was done to factor out the inherent differences in language, and isolate the performance improvement given by the RCU algorithm. This is important because ultimately, the decision on whether to

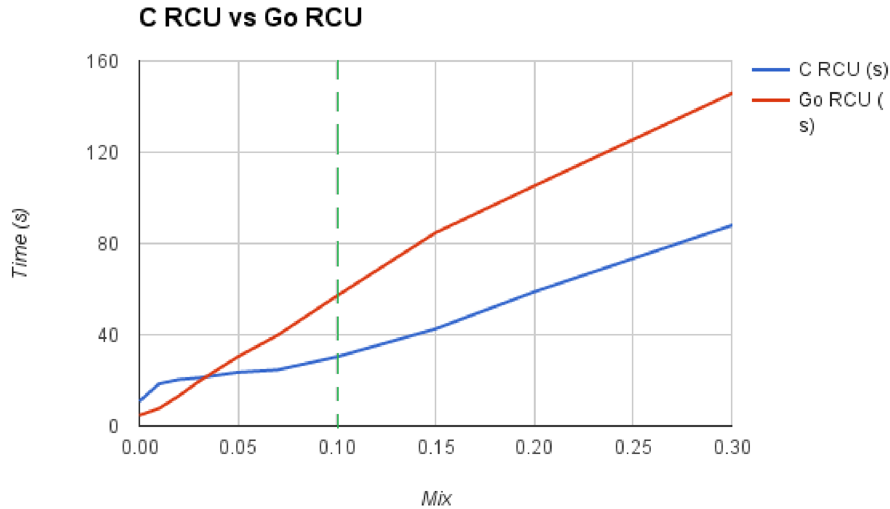

Figure 9: Direct comparison of times taken by Go RCU and C RCU to perform the benchmark operation, for various mixes. Mixes up to 30% are shown for the sake of displaying all data, but data for mixes of greater than 10% are not meaningful since RCU would almost never be used in those situations.

use RCU in a particular language will be driven by its improvement over the next best alternative (RW mutex), rather than by its absolute performance. The relevant metric in this benchmark was *percentage performance improvement over RW mutex*. For every mix, the percentage improvement in speed of RCU over the corresponding RW mutex linked list was calculated. A plot of these improvements is shown in fig. 10.

Go RCU provides a greater speedup over RW mutexes than C RCU for mixes of less than about 3%. Interestingly, the threshold is almost identical to that in the absolute performance comparison. For a mix of 3%, Go RCU offered a speedup of 73.66%, while C RCU had a speedup of 75.45%. For mixes of less than 1%, Go RCU had massive speedups of almost 90% (about 10x faster) over RW mutexes, while C RCU achieved a max speedup 77.04% for a 1% mix. Go RCU provided a positive performance improvement for mixes of less than about 15%, while C RCU would be beneficial in linked lists for mixes of less than about 22% (based on an interpolation of the line).

3.1.3 Garbage Collector Activity

In order to ensure that Go's performance (especially at the lower mixes where it outperforms C) was not simply the result of it doing very limited garbage collection (trading speed for memory), the number of garbage collections was tracked for all mixes. The results are shown in fig. 11.

The number of garbage collections increased linearly with the mix. This is reasonable because in RCU, memory only needs to be freed after write operations, so it is expected that the number of garbage collections would increase in lockstep with the number of writes. This also shows that Go's performance includes the overhead of garbage collection.

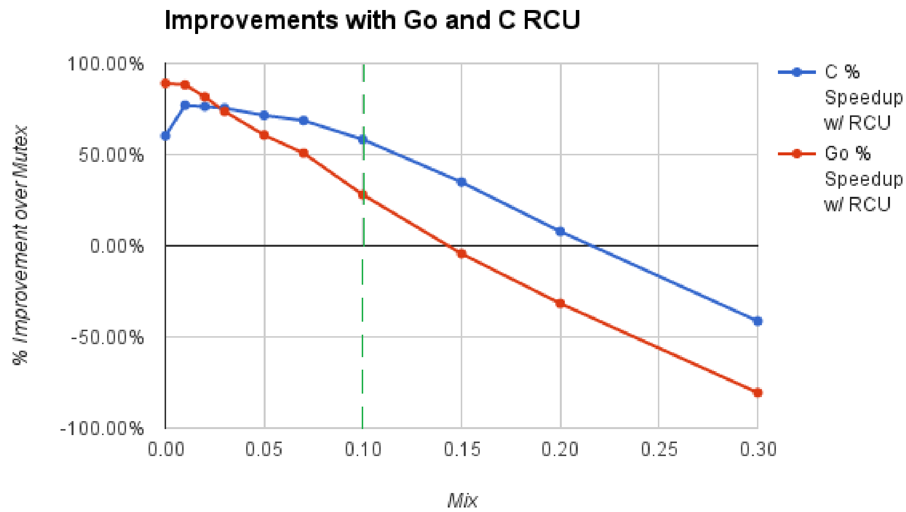

Figure 10: The percent performance improvement over RW mutex for Go RCU and C RCU. Again, mixes up to 30% are shown for the sake of displaying all data, but anything beyond 10% is not particularly meaningful.

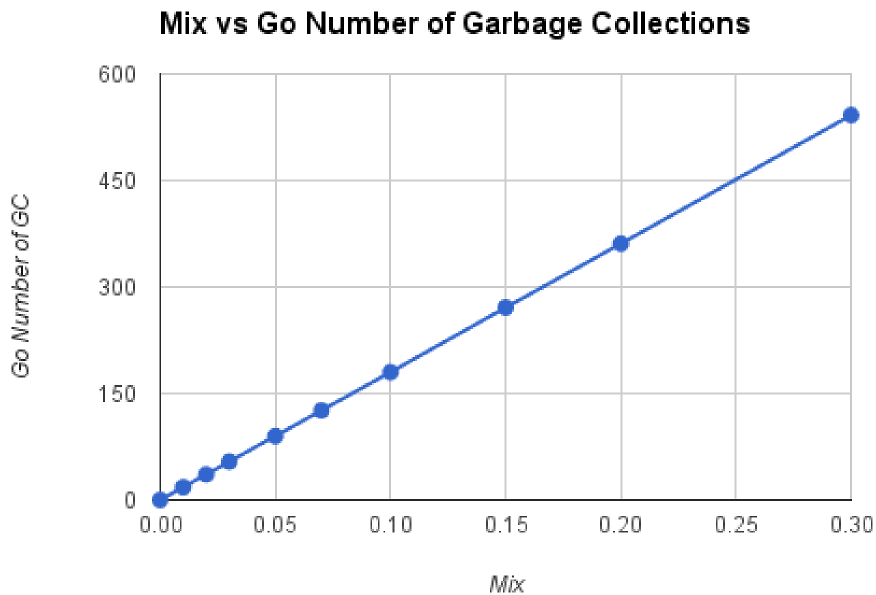

Figure 11: The mix versus the number of garbage collections done by Go

Table 2: RCU API Components in C and Go

API Function	C Necessary	Go Necessary
`rcu_read_lock()`	Y	N
`rcu_read_unlock()`	Y	N
`synchronize_rcu()`	Y	N
`call_rcu()`	Y	N
`rcu_assign_pointer()`	Y	Y
`rcu_dereference()`	Y	Y

3.2 Ease of Use

Go RCU was significantly easier to use than C RCU. This is because significantly fewer functions in the RCU API must be called every time an an operation is performed on RCU-protected data. This is highlighted in table 2.

As mentioned earlier, because of Go's built-in garbage collector, there is no need to detect quiescent states; thus, an application need not call `rcu_read_lock()` and `rcu_read_unlock()`. In addition, since the garbage collector takes care of freeing memory, the application does not need to call `synchronize_rcu()` or `call_rcu()`. These two functions exist to mark when it is safe to free memory, and this is superfluous in a garbage collected language. Of the functions in the original RCU API, only two are necessary in Go RCU.

Go RCU is easier to use not just because of the reduction in the number of function calls, but also because of the reduction in the difficulty of deciding, as an RCU user, where to place these function calls. In C RCU, whenever a reading operation is done, `rcu_read_lock()` must be called just before, and `rcu_read_unlock()` just after. Forgetting one of these calls would derail the quiescent state detection mechanism. On the other hand, as explained before, in Go RCU, the user would not need to worry about this.

Even more complicated from the user's perspective is calling `synchronize_rcu()` or `call_rcu()`. `synchronize_rcu()` must be called immediately after the pointers have been atomically switched. It will then block the thread until the grace period is over, at which point the code to free the memory must follow the function call. This step of calling `synchronize_rcu()` and then freeing memory must be followed every time the RCU-protected data structure is modified. Since `call_rcu()` is simply a callback form of the same function, it encounters similar issues: it must be called with every modification, except that the memory reclamation code is inside the called function as opposed to after a `synchronize_rcu()` call.

This sequence of function calls can make implementing RCU in C a laborious and error-prone process. We encountered many bugs in the development of the C RCU linked list for this paper due to the misplacement of function calls. Go RCU avoids these issues, and is therefore much simpler to utilize in an application.

4 Conclusions

As the number of cores on computers increase, it has become increasingly important to create parallelized programs that can take advantage of these multicore resources. Parallel programs, however, run into problems with synchronizing access to shared memory.

While atomic operations and locking can sometimes provide effective solutions, RCU is particularly efficient for situations in which the writer-reader ratio, or mix, is low. RCU is most

commonly used in the Linux kernel, which is written in C. In C, RCU is relatively difficult to use and bug-prone. It also ends up implementing much of the functionality of a garbage collector. This paper, therefore, tested the viability of RCU in a language that already supports garbage collection. Specifically, it investigates if RCU in Go would produce code that is easier to use, while not adversely affecting performance.

Based on the results stated in this paper, it appears that RCU is viable in a garbage collected language. Implementing RCU in Go allows for a significant reduction in the number of functions in the RCU API. This makes it easier to use and reduces the chances of bugs in the user application. Go RCU's performance is also comparable, and in some situations, even better, than that of C RCU. For very high read loads, Go RCU outperformed C RCU both in absolute performance and in performance relative to RW mutexes.

However, there are possible issues and areas for improvement in these results. Firstly, *userspace-rcu*, the library against which Go RCU was benchmarked, is most probably not as performance-optimized as the RCU implementation in the Linux kernel. Thus, if Go RCU could be benchmarked against Linux's RCU implementation, the results would likely differ. Secondly, in this paper, RCU was implemented only on a linked list. The performance would likely vary with different data structures, such as binary trees, doubly linked lists, etc., so this research is far from comprehensive.

One additional advantage of RCU in garbage collected languages that was discovered through this experimentation was that RCU readers were allowed to sleep or block while inside read-side critical sections. This is not possible with RCU in C, simply because of the contract that the quiescent state detection algorithms require. The Go garbage collector has no such requirements. Thus, some applications that require functionality such as blocking during read side critical sections, would be able to use RCU in a garbage collected environment even if they were not able to in C.

Overall, this experimentation highlights the viability of RCU in a garbage collected language, and opens the doors to further research in this area.

4.1 Future Work

- Perform additional CPU and memory profiling in order to better understand the bottlenecks in Go RCU, and perhaps further improve the performance of RCU in garbage collected languages.

- Implement and test Go RCU on data structures other than linked lists, such as binary trees, sets, doubly linked lists, etc. to see how ease of use and performance differ.

- Implement Go RCU in an OS kernel. Go would bring several benefits to OS RCU beyond the benefits in user space. For example, with Linux RCU, OS threads cannot sleep since threads sleeping would extend grace periods and create inefficiency. If Go RCU were to be used, this would not be a problem.

- There are many applications written in Go that have extremely high read loads, but still use locking. For example, many caches simply lock out all readers while performing updates to the cache data, thus greatly slowing the program and preventing real-time cache response. This research could be continued by integrating RCU into these applications, both improving their performance and supplementing our understanding of RCU performance under real-world loads.

5 Acknowledgements

We would like to thank the MIT PRIMES program for making this research possible, and in particular, we would like to thank our mentor Cody Cutler and Professor Frans Kaashoek for their continued advising and guidance. We would also like to thank our parents for their constant support and encouragement throughout the program.

References

[1] P. E. McKenney, "What is RCU? Part 2: Usage," Dec. 24, 2007. [Online]. Available: https://lwn.net/Articles/263130/

[2] A. Clements, "The scalable commutativity rule: Designing scalable software for multicore processors," Ph.D. dissertation, Massachusetts Institute of Technology, Jun. 2014. [Online]. Available: https://pdos.csail.mit.edu/papers/aclements-phd.pdf

[3] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, Mar. 1, 2005. [Online]. Available: http://www.drdobbs.com/web-development/a-fundamental-turn-toward-concurrency-in/184405990

[4] L. Ceze. (2007, Nov. 28) Atomic operations. Comp. Sci. and Eng., U of Washington. Despite this source's title, it also includes information about locking. [Online]. Available: https://courses.cs.washington.edu/courses/cse378/07au/lectures/L25-Atomic-Operations.pdf

[5] *Threading Programming Guide*, Online Documentation, Apple Inc., Jul. 15, 2014. [Online]. Available: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/Multithreading/ThreadSafety/ThreadSafety.html

[6] P. E. McKenney and J. Walpole, "What is RCU, fundamentally?" Dec. 17, 2007. [Online]. Available: https://lwn.net/Articles/262464/

[7] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 375–382, 2012.

[8] P. E. McKenney, "RCU linux usage," Jan. 11, 2016. [Online]. Available: http://www.rdrop.com/~paulmck/RCU/linuxusage.html

[9] P. E. McKenney, *What is RCU?*, Linux Kernel Organization, Mar. 2016. [Online]. Available: https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt

[10] C. Casteyde, *Bug 15781 - invoked rcu_dereference_check() without protection in net/netfilter/nf_log.c:55*, Linux Bug, Apr. 13, 2010. [Online]. Available: https://bugzilla.kernel.org/show_bug.cgi?id=15781

[11] A. Shah, *Bug 38692 - block/cfq-iosched.c:2776 invoked rcu_dereference_check() without protection!*, Linux Bug, Jul. 2, 2011. [Online]. Available: https://bugzilla.kernel.org/show_bug.cgi?id=38692

[12] R. Hudson, "Go GC: Prioritizing low latency and simplicity," Aug. 31, 2015. [Online]. Available: https://blog.golang.org/go15gc

[13] R. Love, "Lowering latency in Linux: Introducing a preemptible kernel," *Linux Journal*, no. 97, May 1, 2002. [Online]. Available: http://www.linuxjournal.com/article/5600

[14] M. Desnoyers, "Low-impact operating system tracing," Ph.D. dissertation, University of Montreal, Dec. 2009. [Online]. Available: https://publications.polymtl.ca/206/1/2009_MathieuDesnoyers.pdf

[15] P. E. McKenney, *Review Checklist for RCU Patches*, Linux Kernel Organization, Apr. 2014. [Online]. Available: https://www.kernel.org/doc/Documentation/RCU/checklist.txt