Read-Copy Update in a Garbage Collected Environment

By Harshal Sheth, Aashish Welling, and Nihar Sheth
Overview

- Read-copy update (RCU)
  - Synchronization mechanism used in the Linux kernel
  - Mainly used in lower level languages such as C or C++
- Explored the viability of RCU in a garbage collected language: Go
- Go RCU provides similar performance to C++ RCU
- Code simpler and less error-prone in Go RCU
Outline

- Problem
- RCU Background
- Experiment Design
- Results
- Conclusions
- Future Work
- Acknowledgements
Introduction

- Clock speeds are no longer increasing exponentially
Introduction

- Clock speeds are no longer increasing exponentially
- Computers have more cores
- Parallelization is becoming increasingly important
Unprotected Data Access: Initial List

Element A
0100010

Element B
10010111

Element C
00001111
Unprotected Data Access: Write Starts

Element A
0100010

Element B
11000111

Element C
00001111
Unprotected Data Access: Read Occurs

Element A
0100010

Element B
11000111

Element C
00001111

Read 11000111
Unprotected Data Access: Write Finishes

- The reader has read a corrupted value from the list
- This could make the program crash
Synchronizing Parallel Processes

- Multithreaded programs require synchronization
- Many different mechanisms to achieve such synchronization
Read-Write Mutexes

- Mutexes are the conventional method of synchronization
- "Locks" to prevent unsafe concurrent access to memory
- Writing and reading threads cannot operate concurrently
Write Lock

Element A → Element B → Element C
Read Lock

Element A → Element B → Element C
Problem: Locks Limit Scalability

- Ideally, performance should increase linearly with the number of cores.
- If there is high contention, threads are essentially serialized.

From Paul McKenney’s dissertation
Read-Copy Update
Basic RCU Properties

- Prevents data corruption
- Never blocks readers
- Writers are still serialized and have higher overhead
- Good for high reading thread to writing thread ratios
  - This happens a lot in the Linux kernel
RCU Use in Linux Kernel

- Used commonly in Linux kernel and normally implemented in C
- Linux is used everywhere
  - Android
  - Servers
  - etc.

From http://www.rdrop.com/~paulmck/RCU/linuxusage/linux-RCU.png
Example: Initial Linked List
Example: Copy Element

Element A → Element B → Element C

New Element B
Example: Update List Atomically
Example: All Previous Readers Finish
Example: Free Old Element
When Can We Free Memory?

- **Quiescent state**: any time period during which a thread is not reading.
- **Grace period**: time it takes for all threads to go through at least one quiescent state.

http://lwn.net/Articles/323929/
RCU in the Linux Kernel

- Linux kernel written in C
- No garbage collector in C
  - Old copies need to be manually freed
  - Need to wait until a grace period has passed until freeing
  - Difficulty of implementation leads to bugs
  - For example, a recent Linux kernel bug (#102291) dealt with RCU accidentally taking a write lock during a read-side critical section
    - Avoiding bugs is very important in widely used systems
- “RCU is a poor man’s garbage collector”
  - Paul E. McKenney, Inventor of RCU
Our Idea: RCU in a Garbage Collected Language

- Why make a “poor man’s garbage collector” when a full garbage collector is available?
- Garbage collection makes usage significantly easier
  - Garbage collector automatically decides when to free memory - no need to keep track of grace periods manually!
  - Bug 102291 would be avoided in GC environment
- Decided to use Go
  - Designed by Google
Why Go?

● For system-level programming
  ○ Could be used to write a kernel

● Good garbage collector
  ○ Is it good enough?
Experiment Design
Goals

● Is RCU in a garbage collected language a viable option?
  a. Is it easier to implement and/or use?
  b. Does it provide performance benefits similar to RCU in manual memory management languages?
Our Approach

- Implemented RCU in Go
- Compared amount of code that had to be written
- Compared RCU performance in Go to performance in C++
We vary the number of operations that are writes. The % writes is the mix. We used mixes up to 30%.
Results
Go RCU is Indeed Simpler

<table>
<thead>
<tr>
<th>API Function</th>
<th>C++ Necessary</th>
<th>Go Necessary</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcu_read_lock()</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>rcu_read_unlock()</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>synchronize_rcu()</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>call_rcu()</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>rcu_assign_pointer()</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>rcu_dereference()</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

- Programmers are likely to write fewer bugs since it is simpler
Performance of C++ RCU vs. Go RCU

RCU in C++ and Go

- C++ RCU (s)
- Go RCU (s)

Time (seconds)

Mix
Garbage Collection Counts

![Graph: Go Number of Garbage Collections vs Mix]

- **Y-axis:** Go Number of GC
- **X-axis:** Mix
- The graph shows a linear relationship between the Go Number of Garbage Collections and the Mix value, with an increase in Mix leading to an increase in the number of Garbage Collections.
Factoring Out the Programming Language

- Benchmark has RCU portions and non-RCU portions
  - Need to focus on RCU portion
Evaluating RCU

Go RCU
- garbage collected

C++ RCU
- manual memory mgmt.
- userspace-rcu: grace period

Go Mutex
- goroutine
- RW mutex

C++ Mutex
- pthread
- RW mutex

Benchmarked each implementation with same test parameters
Speedups over RW Mutex
Conclusions
Conclusions

- RCU in a garbage collected environment is promising
- Performance improvement vs. RW mutex is similar if not better than improvement in C++
- Don’t need to worry about freeing old copies because of garbage collector
  - Many functions simply not necessary
  - Fewer opportunities for bugs
Future Work

- Integrate Go RCU into an actual application (i.e. cache) to see its real-world performance
- Use Go RCU inside an OS kernel to see how it would perform in kernel space
Acknowledgements

We would like to thank:

- Our mentor Cody Cutler for his guidance and insight
- Prof. Frans Kaashoek for suggesting this project
- Our parents for their constant support and encouragement
- The MIT PRIMES program for making this research possible