Concurrent and Scalable Hash Table for the Go Programming Language

By: Louis Jenkins
Advisor: Michael Spear
What is the Go programming language?

• Concurrent Programming Language
  • Designed to make concurrency easy
  • NOT a parallel programming language

• Philosophy
  • “Do not communicate by sharing memory, instead share memory by communicating”
  • Favors message passing over shared memory synchronization
    • Channels over Locks and Atomics
    • Atomics were only added after Go’s 1.0 Release

• Created, Developed, and Maintained by Google
• Utilizes Cooperative and Preemptive Multitasking through Goroutines
Problems with the Go programming language?

• Data structures are not naturally concurrent-safe
  • Unsynchronized writes can trigger undefined behavior
    • Only one writer is allowed access at any given time
  • Only option is to use coarser-grained synchronization or message passing

• Does NOT allow the user to ‘roll their own’ map
  • There is no safe nor reliable way to hash key objects outside of runtime
  • No way to implement a custom iterator
    • I.E, no for-range looping over your own data structure

• Attempts to create a concurrent map by others in the past fall short
  • Forced to use the built-in map, and restrict the types of keys allowed.
  • Inefficient
Solution: Adding a Concurrent Hash Table

• Contributions
  • Allow Concurrent Reads and Writes
    • Allows true parallelism that Go currently lacks
  • Ensure all access and assignments to the map are atomic
    • No race conditions, utilize fine-grained locking
  • Support all operations that the normal Go hash table has
    • Insertion, Lookup, Remove, Iteration
  • Provide reasonable semantics to the programmer
    • Atomic read-modify-write of single element
    • Guarantees on key/value relationships during iteration
Go: Syntax and Compiler Transformation

Race-Free Map Accesses

- Map accesses not designed to be concurrent
  - Race-Condition where pointer to element returned from map may be mutated or deleted before assignment

- Solution
  - Hold onto the lock until after the assignment
  - Requires a call to ‘maprelease’
    - Added as injected call by compiler
    - Transparent to user
Go: Syntax and Compiler Transformation

AST Injection

. CALLFUNC-init
  .  AS u(100) l(6) tc(1)
  .  .  NAME-main.autotmp_1 u(1) a(true) l(6) x(0+0) class(PAUTO) esc(N) tc(1) assigned used(true) int
  .  .  IND u(100) l(6) tc(1) int
  .  .  .  CALLFUNC u(100) l(6) tc(1) nonnil PTR64-*int
  .  .  .  .  NAME-runtime.mapaccess1_fast64 u(1) a(true) x(0+0) class(PFUNC) tc(1) used(true) FUNC-func(*byte, map[int]int, int) *int
  .  .  .  .  CALLFUNC-list
  .  .  .  .  .  AS u(2) l(6) tc(1)
  .  .  .  .  .  .  INDREG-SP a(true) l(6) x(0+0) tc(1) addrtaken runtime.mapType-2 PTR64-*byte
  .  .  .  .  .  .  ADDR u(2) a(true) l(6) tc(1) PTR64-*uint8
  .  .  .  .  .  .  .  NAME-type.map[int]int u(1) a(true) l(4) x(0+0) class(PEXTERN) tc(1) uint8
  .  .  .  .  .  .  AS u(1) l(6) tc(1)
  .  .  .  .  .  .  .  INDREG-SP a(true) l(6) x(8+0) tc(1) addrtaken runtime.hmap·3 MAP-map[int]int
  .  .  .  .  .  .  .  NAME-main.map_ u(1) a(true) l(4) x(0+0) class(PAUTO) f(1) tc(1) used(true) MAP-map[int]int
  .  .  .  .  .  AS u(1) l(6) tc(1)
  .  .  .  .  .  .  .  INDREG-SP a(true) l(6) x(16+0) tc(1) addrtaken runtime.key·4 int
  .  .  .  .  .  .  .  NAME-main.autotmp_0 u(1) a(true) l(6) x(0+0) class(PAUTO) esc(N) tc(1) assigned used(true) int
  .  .  .  CALLFUNC u(100) l(6) tc(1)
  .  .  NAME-runtime.maprelease u(1) a(true) x(0+0) class(PFUNC) tc(1) used(true) FUNC-func()
  .  AS u(2) l(6) colas(true) tc(1)
  .  .  NAME-main.value u(1) a(true) g(3) l(6) x(0+0) class(PAUTO) f(1) tc(1) assigned used(true) int
  .  .  NAME-main.autotmp_1 u(1) a(true) l(6) x(0+0) class(PAUTO) esc(N) tc(1) assigned used(true) int
New Keyword: ‘sync.Interlocked’ and their Contexts

**Region**
- Allows read-modify-write operations over that key-value pair
  - The bucket’s lock is held for the entire block, allowing for atomic transformations of that data
  - Acts as a per-bucket mutex

**Range**
- Changes semantics of for-range loop over map from per-bucket snapshot, to interlocked bucket iteration.

```c
for key, value := range sync.Interlocked map_ {  // Imagine value is a pointer to a struct  // Guaranteed to be safe and without race conditions  // so long as the sync.Interlocked keyword is used.    value.field++  }
```
Iteration – Snapshot and Interlocked

**Atomic Snapshot**

- Lock -> Copy -> Unlock -> Iterate Copy
  - Iterate over snapshot of bucket data
    - More fine-grained than interlocked
    - Does not guarantee consistency
      - I.E: Element was removed from map, but is present in the snapshot
      - Garbage Collector does not collect the pointer data in the snapshot until it finishes
    - Allows blocking and/or long operations to process data without stalling other Goroutines.
  - Moderate Overhead

**Interlocked**

- Lock -> Iterate -> Unlock
  - Iterate over actual bucket data
    - More coarse-grained than atomic snapshot
    - Guarantees the key and value are only accessible by the current Goroutine
      - ‘map[key] == value’ always true
    - While iterating, no other Goroutine may access that data, so long operations stall other Goroutines
  - No overhead

**Both**

- Allow concurrent and parallel readers and writers, as well as iterators
- High scalability, although slower than the built-in iterator
Concurrent Map Design

Legend:

- Ø → Empty Bucket (nil)
- ∈ → Non-Empty Bucket (Not Locked)
- ↩ → Recursive Bucket
-🔒 → Non-Empty Bucket (Locked)

- Per-Bucket Locking
  - Must be acquired by contending Goroutines before accessing
  - Utilizes a Test-And-Test-And-Set Spinlock with dynamic yielding and exponential back-off under contention
  - Fine-grained locking

- Resizing
  - When a bucket is full, the bucket’s data will be rehashed to another array of buckets. The bucket’s descriptor will be updated to point to the aforementioned array of buckets. This is referred to as a ‘Recursive Bucket’
  - The new array of buckets are twice the size of the previous, and use a different seed to hash which reduces collision as well as possible contention

- Deadlock Prevention
  - Only one bucket descriptor lock may be held by any given Goroutine at a time.
Concurrent Map Design

**Insertion**

- Insert
- Acquire lock on bucket
- Finish Insertion

**Resize**

- Insert
- Acquire lock on bucket
- Full, Rehash to new array and update descriptor
Concurrent Map Design

Lookup

Want this

Acquired Lock
Concurrent Map Design

Iteration – Atomic Snapshot

Fixed Start

Randomized Start

Iteration - Interlocked
## Benchmark – Integer Set

**MICROBENCHMARK - INTSET (8-CORE)**

<table>
<thead>
<tr>
<th>GOROUTINES</th>
<th>ConcurrentMap</th>
<th>SynchronizedMap (Mutex)</th>
<th>ReaderWriterMap (RWMutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.24</td>
<td>5.13</td>
<td>4.29</td>
</tr>
<tr>
<td>2</td>
<td>6.54</td>
<td>3.72</td>
<td>2.14</td>
</tr>
<tr>
<td>4</td>
<td>11.9</td>
<td>3.24</td>
<td>2.35</td>
</tr>
<tr>
<td>8</td>
<td>22.22</td>
<td>2.75</td>
<td>2.47</td>
</tr>
<tr>
<td>16</td>
<td>19.23</td>
<td>2.95</td>
<td>1.98</td>
</tr>
<tr>
<td>32</td>
<td>22.22</td>
<td>2.85</td>
<td>2.17</td>
</tr>
</tbody>
</table>

**MILLION OPS/SEC**

![Graph showing comparison of ConcurrentMap, SynchronizedMap (Mutex), and ReaderWriterMap (RWMutex) across different numbers of goroutines.](image)
Benchmark – Integer Set

MICROBENCHMARK - INTSET (24-CORE)

<table>
<thead>
<tr>
<th>Goroutines</th>
<th>ConcurrentMap</th>
<th>SynchronizedMap (Mutex)</th>
<th>ReaderWriterMap (RWMutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.57</td>
<td>5.18</td>
<td>4.95</td>
</tr>
<tr>
<td>2</td>
<td>8.13</td>
<td>4.93</td>
<td>3.08</td>
</tr>
<tr>
<td>4</td>
<td>13.33</td>
<td>2.59</td>
<td>2.92</td>
</tr>
<tr>
<td>8</td>
<td>27.03</td>
<td>1.67</td>
<td>2.9</td>
</tr>
<tr>
<td>16</td>
<td>41.67</td>
<td>1.81</td>
<td>2.67</td>
</tr>
<tr>
<td>32</td>
<td>62.5</td>
<td>1.51</td>
<td>2.59</td>
</tr>
</tbody>
</table>
Benchmark – Read-Only Iteration

**MICROBENCHMARK - READ-ONLY ITERATION (8-CORE)**

<table>
<thead>
<tr>
<th>GOROUTINES</th>
<th>ConcurrentMap</th>
<th>ConcurrentMap-Interlocked</th>
<th>DefaultMap (No Lock)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.87</td>
<td>17.86</td>
<td>33.33</td>
</tr>
<tr>
<td>2</td>
<td>13.7</td>
<td>15.62</td>
<td>28.57</td>
</tr>
<tr>
<td>4</td>
<td>14.08</td>
<td>15.38</td>
<td>23.26</td>
</tr>
<tr>
<td>8</td>
<td>5.05</td>
<td>7.81</td>
<td>11.76</td>
</tr>
<tr>
<td>16</td>
<td>3.79</td>
<td>3.85</td>
<td>7.52</td>
</tr>
<tr>
<td>32</td>
<td>2.45</td>
<td>2.28</td>
<td>3.97</td>
</tr>
</tbody>
</table>
Benchmark – Read-Write Iteration

<table>
<thead>
<tr>
<th>Goroutines</th>
<th>ConcurrentMap</th>
<th>ConcurrentMap-Interlocked</th>
<th>SynchronizedMap (Mutex)</th>
<th>ReaderWriterMap (RWMutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.6</td>
<td>3.26</td>
<td>10</td>
<td>10.1</td>
</tr>
<tr>
<td>2</td>
<td>2.49</td>
<td>2.33</td>
<td>4.13</td>
<td>5.65</td>
</tr>
<tr>
<td>4</td>
<td>2.21</td>
<td>2.16</td>
<td>2.45</td>
<td>2.77</td>
</tr>
<tr>
<td>8</td>
<td>1.56</td>
<td>1.59</td>
<td>1.15</td>
<td>1.38</td>
</tr>
<tr>
<td>16</td>
<td>0.81</td>
<td>0.82</td>
<td>0.55</td>
<td>0.68</td>
</tr>
<tr>
<td>32</td>
<td>0.41</td>
<td>0.47</td>
<td>0.33</td>
<td>0.3</td>
</tr>
</tbody>
</table>

MICROBENCHMARK - READ-WRITE ITERATION (8-CORE)
Benchmark – Read-Write Iteration

**MICROBENCHMARK - READ-WRITE ITERATION (24-CORE)**

<table>
<thead>
<tr>
<th>Goroutines</th>
<th>ConcurrentMap</th>
<th>ConcurrentMap-Interlocked</th>
<th>SynchronizedMap (Mutex)</th>
<th>ReaderWriterMap (RWMutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.56</td>
<td>2.9</td>
<td>8.93</td>
<td>8.93</td>
</tr>
<tr>
<td>2</td>
<td>2.25</td>
<td>2.92</td>
<td>4.46</td>
<td>4.48</td>
</tr>
<tr>
<td>4</td>
<td>2.3</td>
<td>2.79</td>
<td>2.23</td>
<td>2.17</td>
</tr>
<tr>
<td>8</td>
<td>2.36</td>
<td>1.78</td>
<td>1.12</td>
<td>1.1</td>
</tr>
<tr>
<td>16</td>
<td>1.52</td>
<td>1.58</td>
<td>0.56</td>
<td>0.55</td>
</tr>
<tr>
<td>32</td>
<td>1.09</td>
<td>1.26</td>
<td>0.3</td>
<td>0.29</td>
</tr>
</tbody>
</table>

MILLION OPS/SEC
Benchmark – Combined Insert, Lookup, Removal, Iteration

**MICROBENCHMARK - COMBINED (8-CORE)**

<table>
<thead>
<tr>
<th>GOROUTINES</th>
<th>ConcurrentMap</th>
<th>ConcurrentMap-Interlocked</th>
<th>SynchronizedMap (Mutex)</th>
<th>ReaderWriterMap (RWMutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.85</td>
<td>6.58</td>
<td>9.71</td>
<td>10.42</td>
</tr>
<tr>
<td>2</td>
<td>10.87</td>
<td>10.87</td>
<td>11.76</td>
<td>10.1</td>
</tr>
<tr>
<td>4</td>
<td>20.83</td>
<td>27.78</td>
<td>12.99</td>
<td>9.62</td>
</tr>
<tr>
<td>8</td>
<td>43.48</td>
<td>47.62</td>
<td>17.24</td>
<td>10.42</td>
</tr>
<tr>
<td>16</td>
<td>55.56</td>
<td>55.56</td>
<td>14.29</td>
<td>10.99</td>
</tr>
<tr>
<td>32</td>
<td>58.82</td>
<td>62.5</td>
<td>13.7</td>
<td>10.53</td>
</tr>
</tbody>
</table>
Benchmark – Combined Insert, Lookup, Removal, Iteration

**MICROBENCHMARK - COMBINED (24-CORE)**

<table>
<thead>
<tr>
<th>Goroutines</th>
<th>ConcurrentMap</th>
<th>ConcurrentMap-Interlocked</th>
<th>SynchronizedMap (Mutex)</th>
<th>ReaderWriterMap (RWMutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.99</td>
<td>5.43</td>
<td>9.09</td>
<td>10.53</td>
</tr>
<tr>
<td>2</td>
<td>14.49</td>
<td>14.71</td>
<td>13.7</td>
<td>9.17</td>
</tr>
<tr>
<td>4</td>
<td>37.04</td>
<td>41.67</td>
<td>18.87</td>
<td>12.5</td>
</tr>
<tr>
<td>8</td>
<td>90.91</td>
<td>76.92</td>
<td>22.22</td>
<td>14.93</td>
</tr>
<tr>
<td>16</td>
<td>142.86</td>
<td>100</td>
<td>21.28</td>
<td>15.38</td>
</tr>
<tr>
<td>32</td>
<td>200</td>
<td>200</td>
<td>23.26</td>
<td>16.13</td>
</tr>
</tbody>
</table>

MILLION OPS/SEC

GOROUTINES
Conclusion

• Only a summary
  • Wasn’t enough time to go into detail about more

• Effort
  • 36 Commits
  • 7,300+ Line of Code additions

• Future
  • Submit proposal to Google’s Go language development team
    • Relatively familiar with mailing list
    • Continue contributing to Go
  • Publish research paper with Dr. Spear
    • Submission to tech conference in a few weeks