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 langauage
- 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

Original Code

```
value := map_[key]
```

Transformation Pseudocode

```
// Temporary used to hold key, Write-barrier protected assignments used
autotmp_1 := key
// Transformed into a function call; Passes compile-time type information 'mapType',
// the header/descriptor 'hmap' of the map, and address of key;
// Returns pointer to value (NOTE: The pointer is pointing INSIDE the map)
autotmp_2 := mapaccess(mapType, hmap, &autotmp_1)
// Create a temporary object to hold the data pointer returned
autotmp_3 := temp(mapType.Value)
// RACE CONDITION: Pointer to value could have been modified or deleted
memcpy(autotmp_2, &autotmp_3, mapType.Value.Size)
value := autotmp_3
```

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) g(1) 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

```
// Assume integer key-value pairs
sync.Interlocked map_[key] {
    map_[key]++ // Remember, this is a RMW operation
}
```

Range

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

```
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

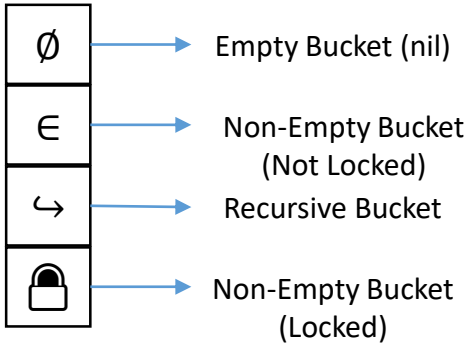
- Lock -> Iterate -> Unlock
  - Iterate over actual bucket data
    - More coarse-grained than atomic snapshot
    - Guarantees the key and value are only accessible by the current Goroutine
      - ‘map[key] == value’ always true
    - While iterating, no other Goroutine may access that data, so long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

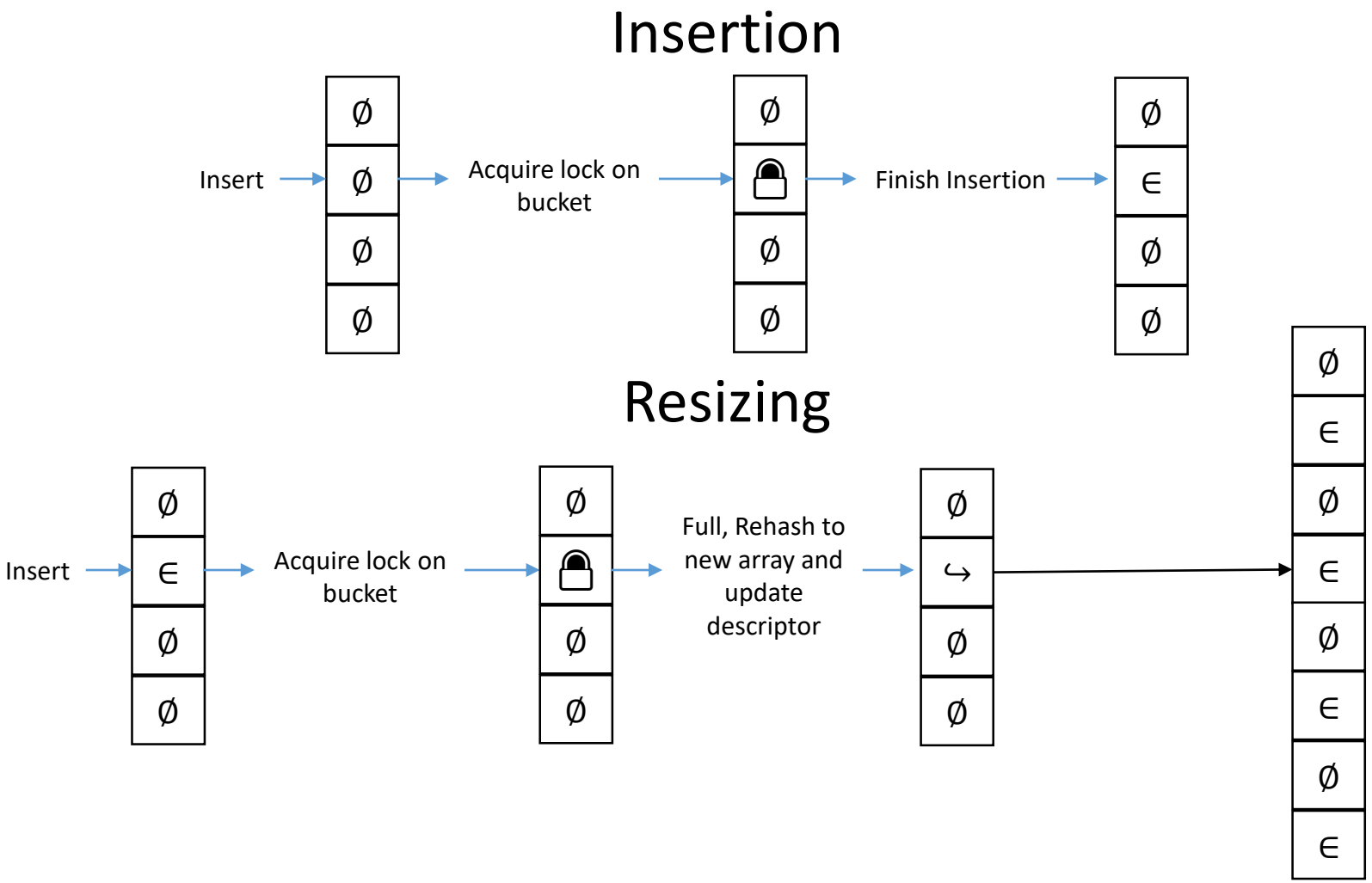
Concurrent Map Design

Legend:



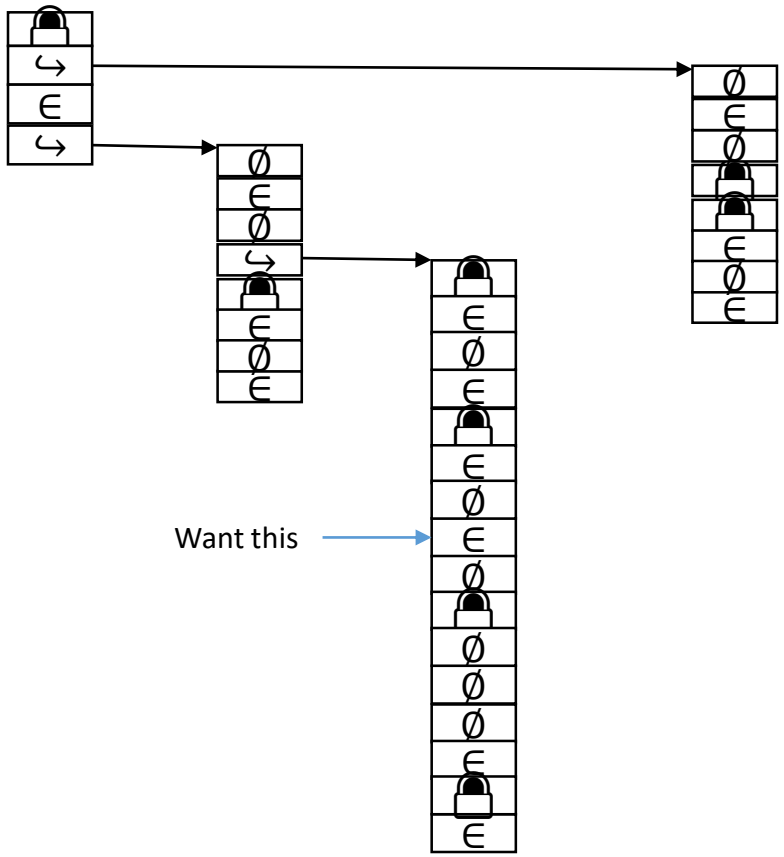
- Per-Bucket Locking
  - Must be acquired by contending Goroutines before accessing
  - Utilizes a Test-And-Test-And-Set Spinlock with dynamic yielding and exponential back-off under contention
  - Fine-grained locking
- Resizing
  - When a bucket is full, the bucket's data will be rehashed to another 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.

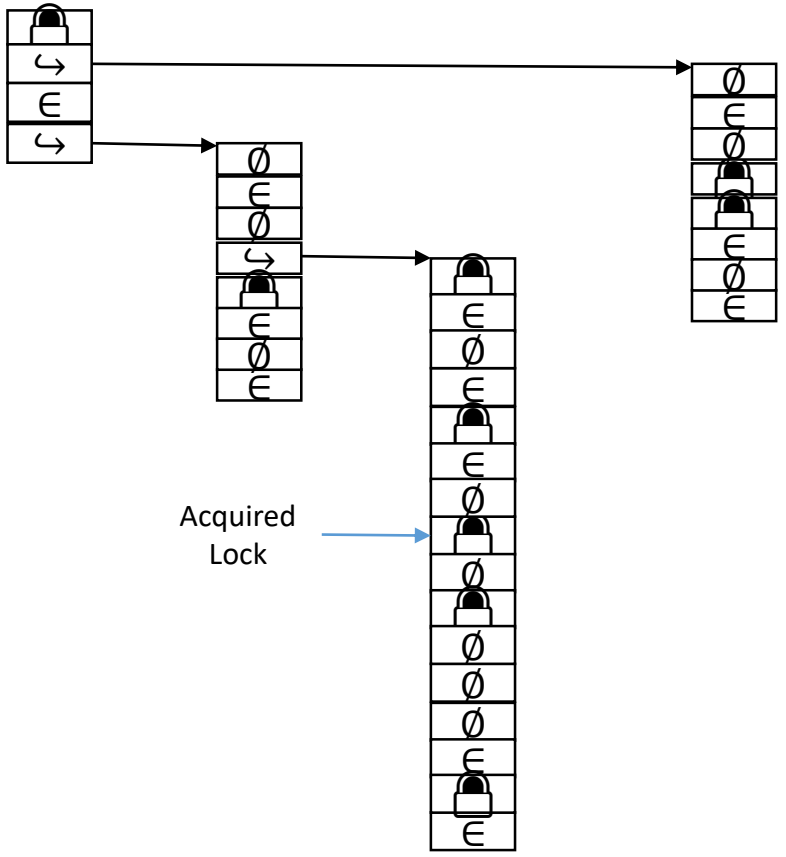
Concurrent Map Design



Concurrent Map Design

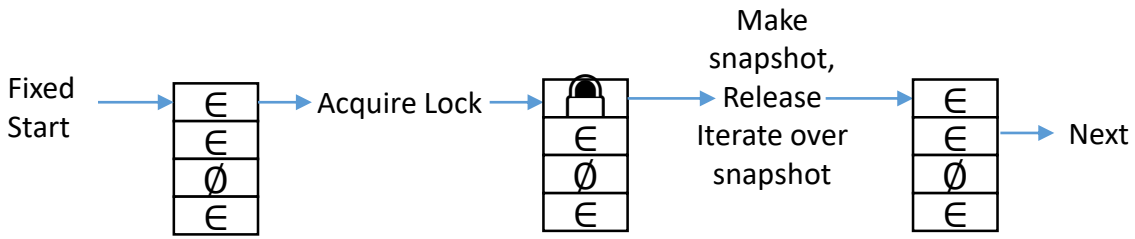
Lookup



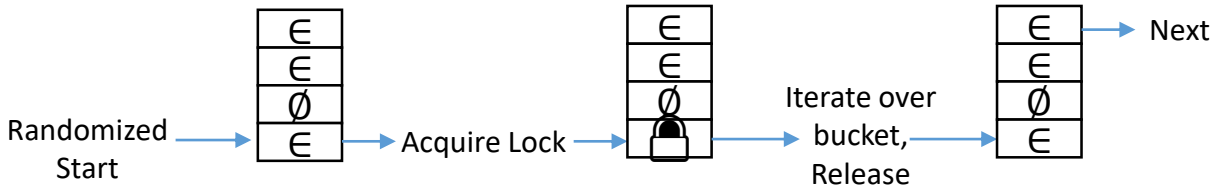


Concurrent Map Design

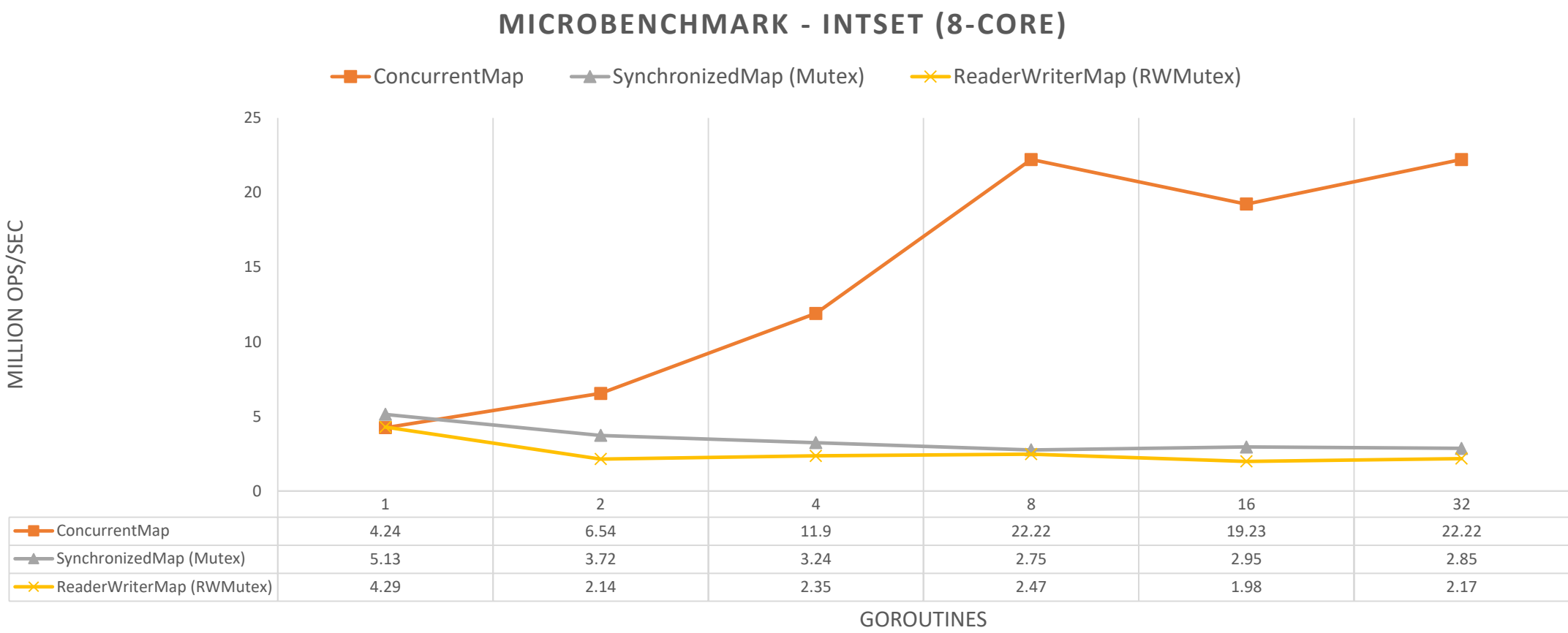
Iteration – Atomic Snapshot



Iteration - Interlocked

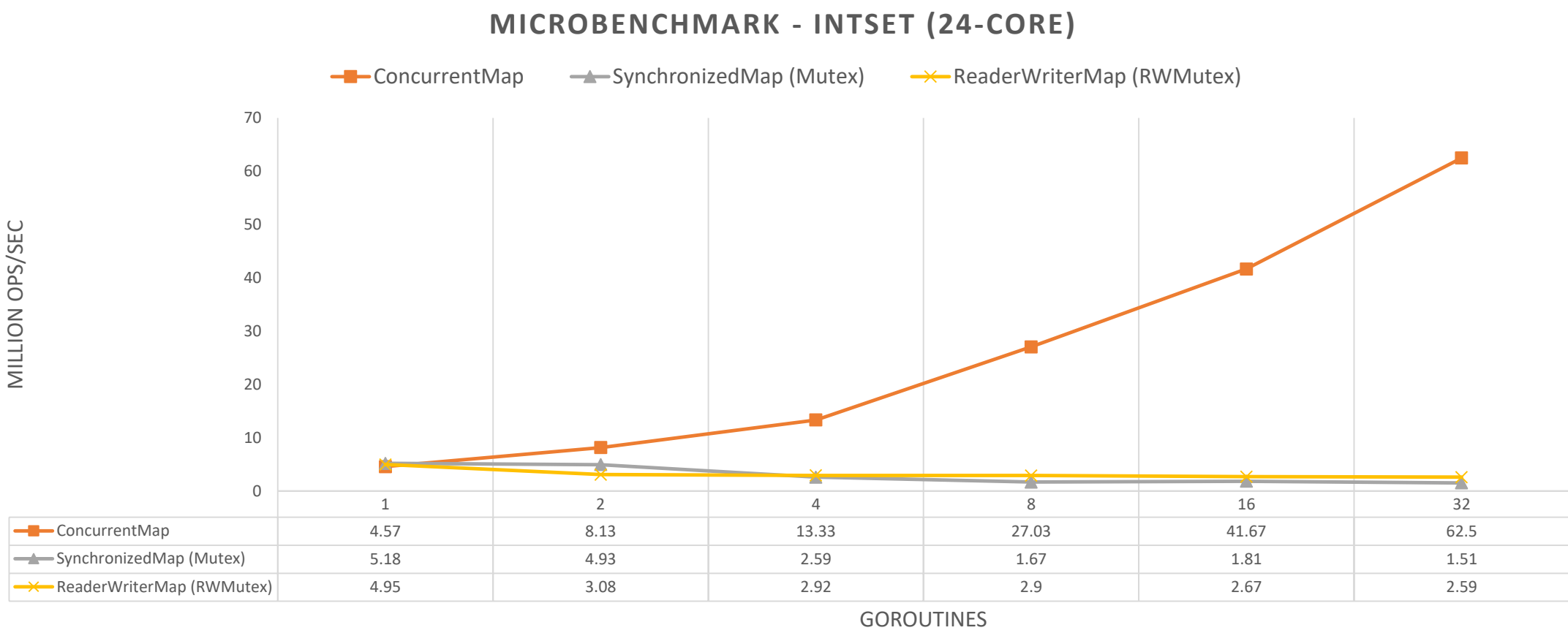


Benchmark – Integer Set



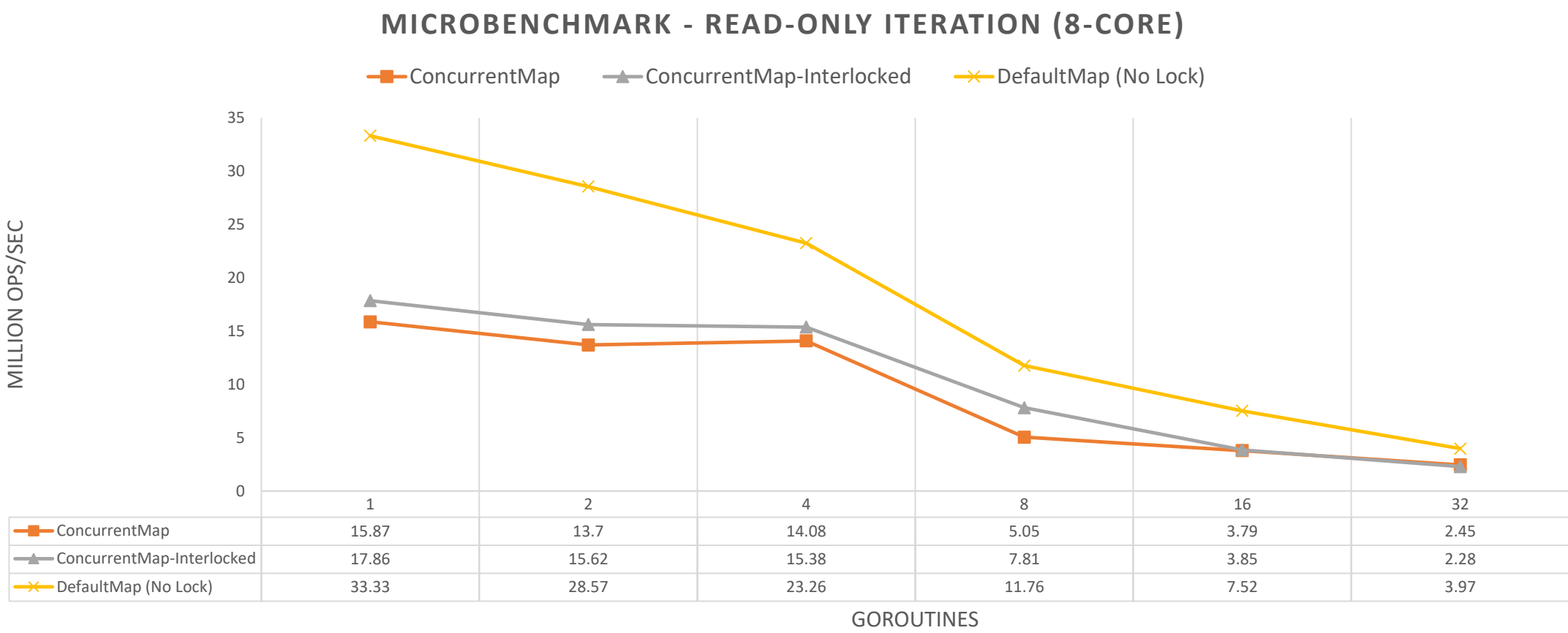
	1	2	4	8	16	32
ConcurrentMap	4.24	6.54	11.9	22.22	19.23	22.22
SynchronizedMap (Mutex)	5.13	3.72	3.24	2.75	2.95	2.85
ReaderWriterMap (RWMutex)	4.29	2.14	2.35	2.47	1.98	2.17

Benchmark – Integer Set



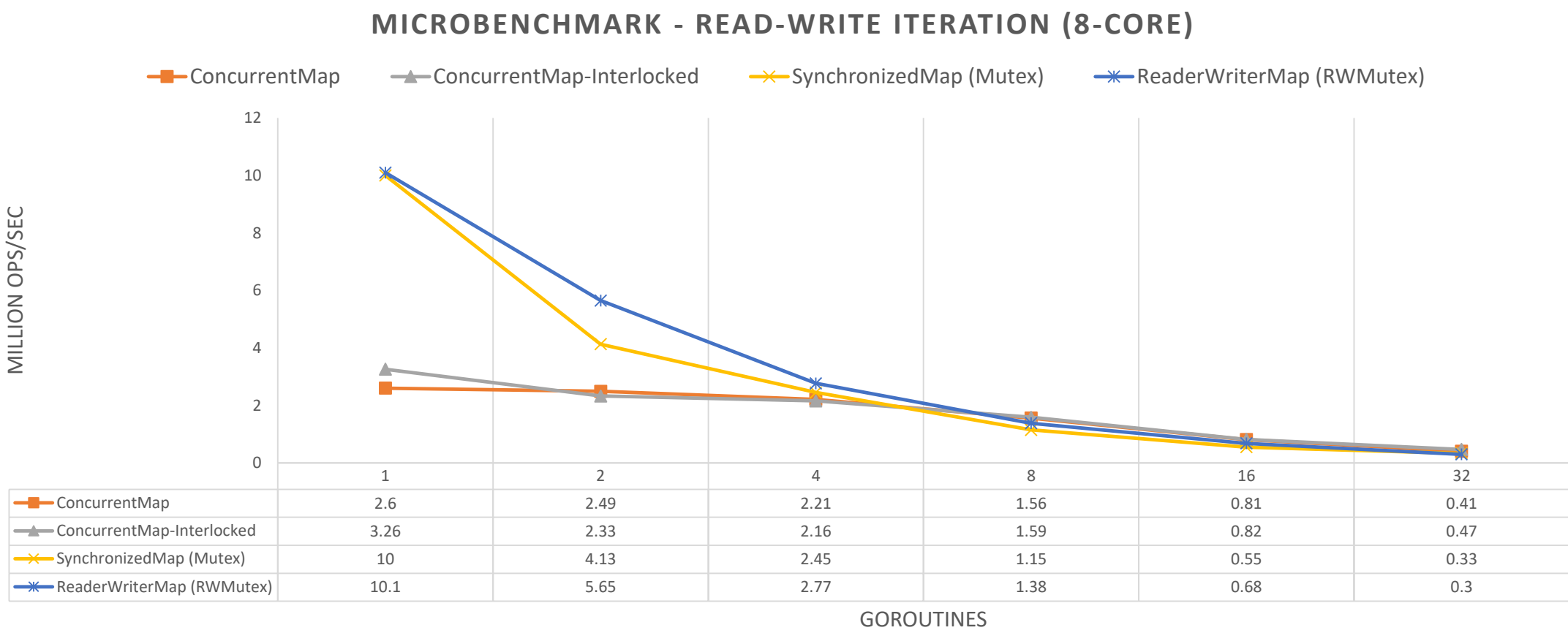
	1	2	4	8	16	32
ConcurrentMap	4.57	8.13	13.33	27.03	41.67	62.5
SynchronizedMap (Mutex)	5.18	4.93	2.59	1.67	1.81	1.51
ReaderWriterMap (RWMutex)	4.95	3.08	2.92	2.9	2.67	2.59

Benchmark – Read-Only Iteration



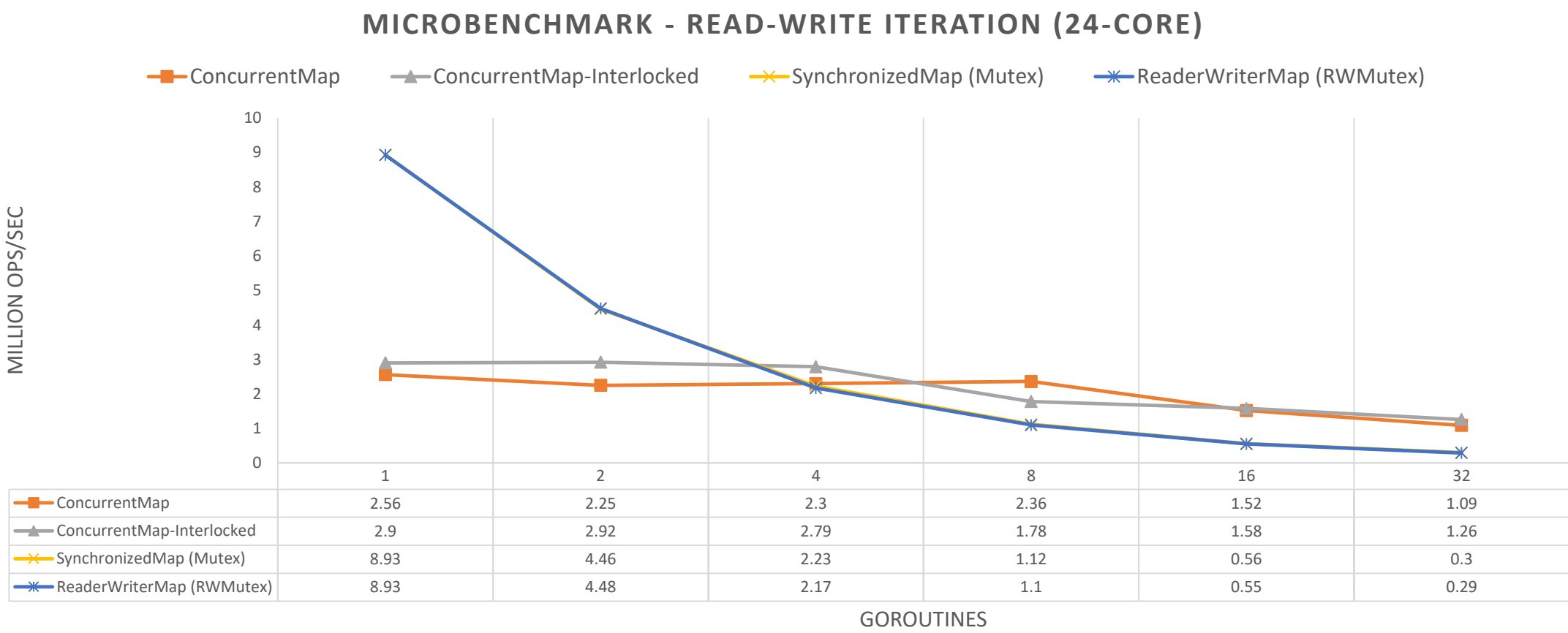
	1	2	4	8	16	32
ConcurrentMap	15.87	13.7	14.08	5.05	3.79	2.45
ConcurrentMap-Interlocked	17.86	15.62	15.38	7.81	3.85	2.28
DefaultMap (No Lock)	33.33	28.57	23.26	11.76	7.52	3.97

Benchmark – Read-Write Iteration



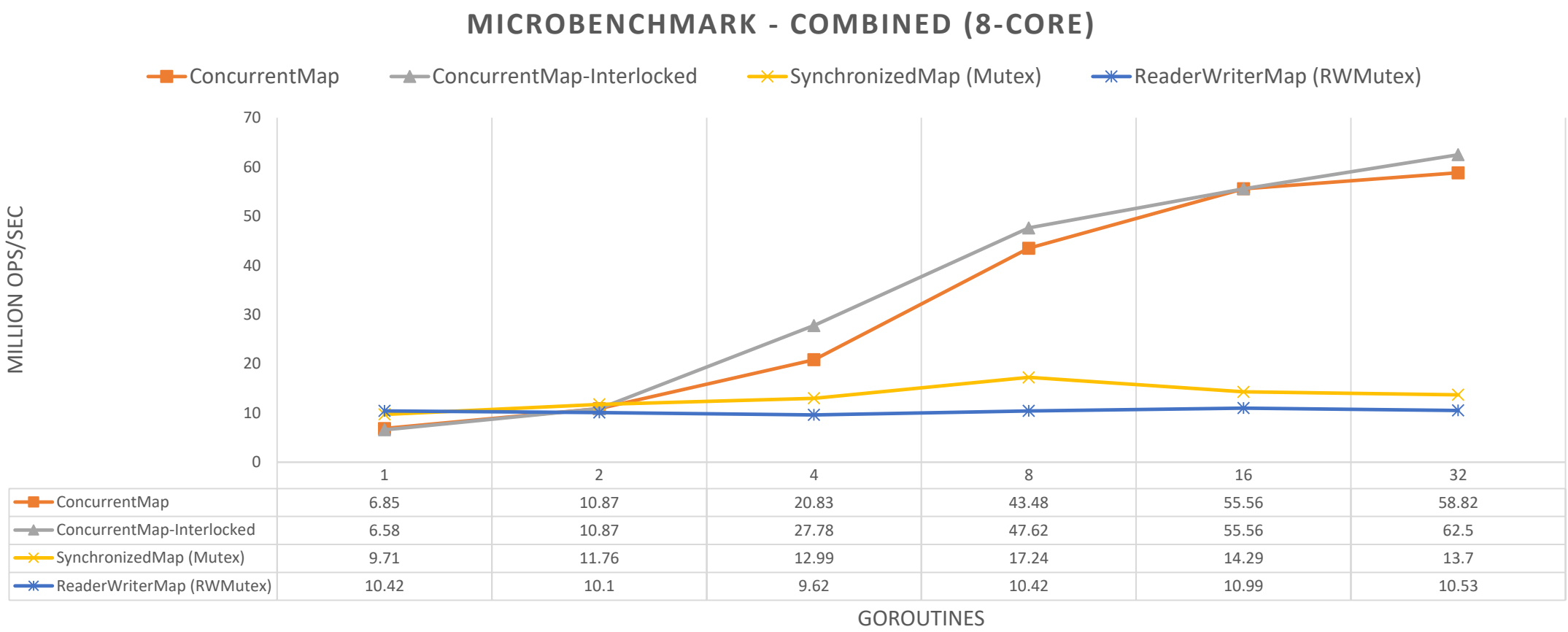
	1	2	4	8	16	32
ConcurrentMap	2.6	2.49	2.21	1.56	0.81	0.41
ConcurrentMap-Interlocked	3.26	2.33	2.16	1.59	0.82	0.47
SynchronizedMap (Mutex)	10	4.13	2.45	1.15	0.55	0.33
ReaderWriterMap (RWMutex)	10.1	5.65	2.77	1.38	0.68	0.3

Benchmark – Read-Write Iteration



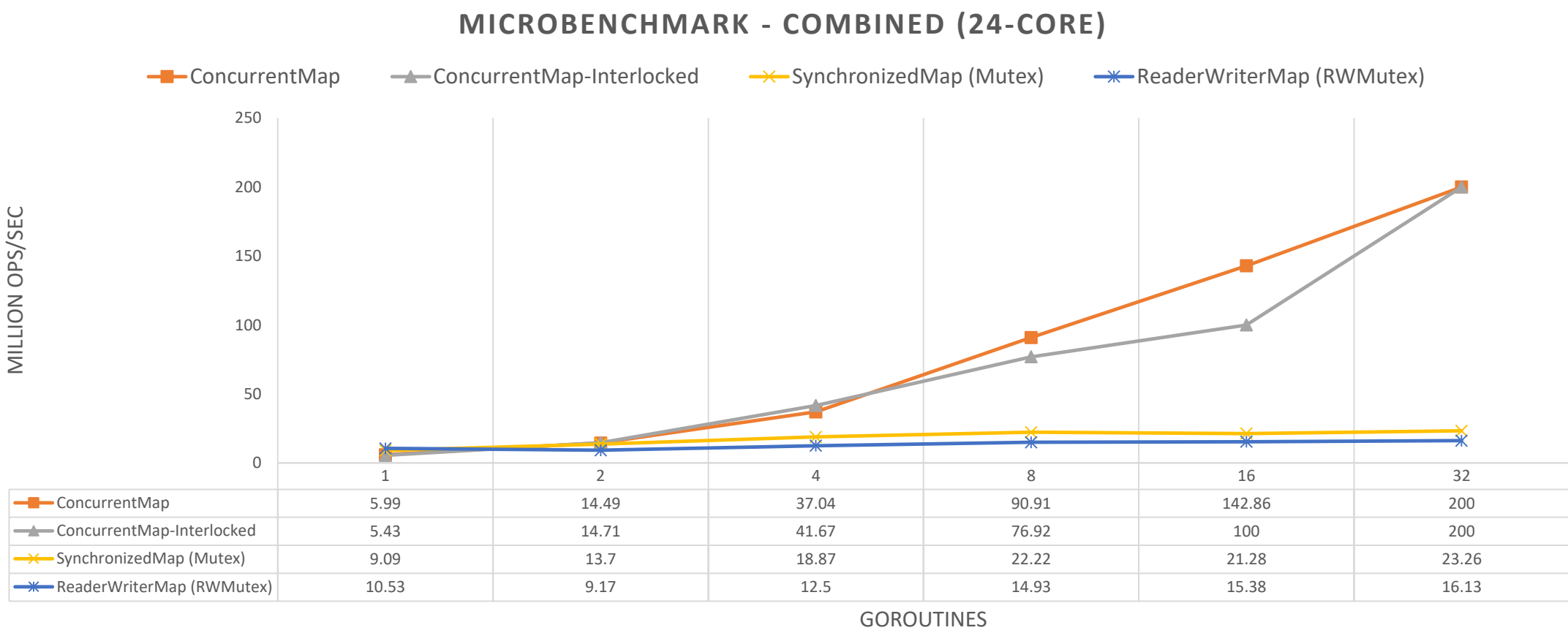
	1	2	4	8	16	32
ConcurrentMap	2.56	2.25	2.3	2.36	1.52	1.09
ConcurrentMap-Interlocked	2.9	2.92	2.79	1.78	1.58	1.26
SynchronizedMap (Mutex)	8.93	4.46	2.23	1.12	0.56	0.3
ReaderWriterMap (RWMutex)	8.93	4.48	2.17	1.1	0.55	0.29

Benchmark – Combined
Insert, Lookup, Removal, Iteration



	1	2	4	8	16	32
ConcurrentMap	6.85	10.87	20.83	43.48	55.56	58.82
ConcurrentMap-Interlocked	6.58	10.87	27.78	47.62	55.56	62.5
SynchronizedMap (Mutex)	9.71	11.76	12.99	17.24	14.29	13.7
ReaderWriterMap (RWMutex)	10.42	10.1	9.62	10.42	10.99	10.53

Benchmark – Combined
Insert, Lookup, Removal, Iteration



	1	2	4	8	16	32
ConcurrentMap	5.99	14.49	37.04	90.91	142.86	200
ConcurrentMap-Interlocked	5.43	14.71	41.67	76.92	100	200
SynchronizedMap (Mutex)	9.09	13.7	18.87	22.22	21.28	23.26
ReaderWriterMap (RWMutex)	10.53	9.17	12.5	14.93	15.38	16.13

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