Comparative Analysis of Concurrency Control Methods in Maps

LAKSHIT SINGH, Hansraj College, University of Delhi, India

Concurrency is a fundamental concern for high-performance systems, and through the years, different methods have emerged that govern how a shared piece of data can be accessed. In this study, I analyze different concurrency control methods to understand their characteristics and impact on performance. We compare 3 different methods, Lock-Free, RWMutex, partially Lock-Free. These implementations are further tested on different workloads, such as Read-Heavy, Write-Heavy and Balanced. Several metrics are taken into account, including throughput, latency, CPU utilization, memory utilization, and more.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**; *Concurrent programming languages*; • **Theory of computation** → Concurrent algorithms; **Concurrency**.

Additional Key Words and Phrases: Concurrency, Concurrent Maps, Lock-Free, RWMutex, Atomic Pointer, sync.Map, Go, Benchmarking, Synchronization, Performance Evaluation, Throughput, Latency, Memory Utilization

ACM Reference Format:
Lakshit Singh. 2025. Comparative Analysis of Concurrency Control Methods in Maps. 1, 1 (August 2025), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Concurrency is essential for modern software systems, allowing applications to perform multiple tasks simultaneously and efficiently utilize multicore processors.

To further enhance performance, many tasks work on a piece of shared data. But when improperly managed, this can lead to race conditions and performance bottlenecks when contention arises. To counter this, different concurrency control methods have emerged, from traditional locking mechanisms to more modern lock-free methods.

My goal with this study is to conduct a systematic benchmarking of various control methods in different situations and realistic loads, providing a detailed analysis of where each method is better or worse.

2 Background And Related work

To enable safe and efficient shared data management in concurrent programs, several synchronization primitives and data structures are used, which include maps, mutexes, atomic operations, and lock-free techniques and more.

2.1 Hashmap (Map)

Maps are a widely used data structures that store data as a key-value pair and typically allow constant-time $O(1)$ access to retrieve a value based on the key.

Author's Contact Information: Lakshit Singh, lakshit.singh.mail@gmail.com, Hansraj College, University of Delhi, New Delhi, Delhi, India.

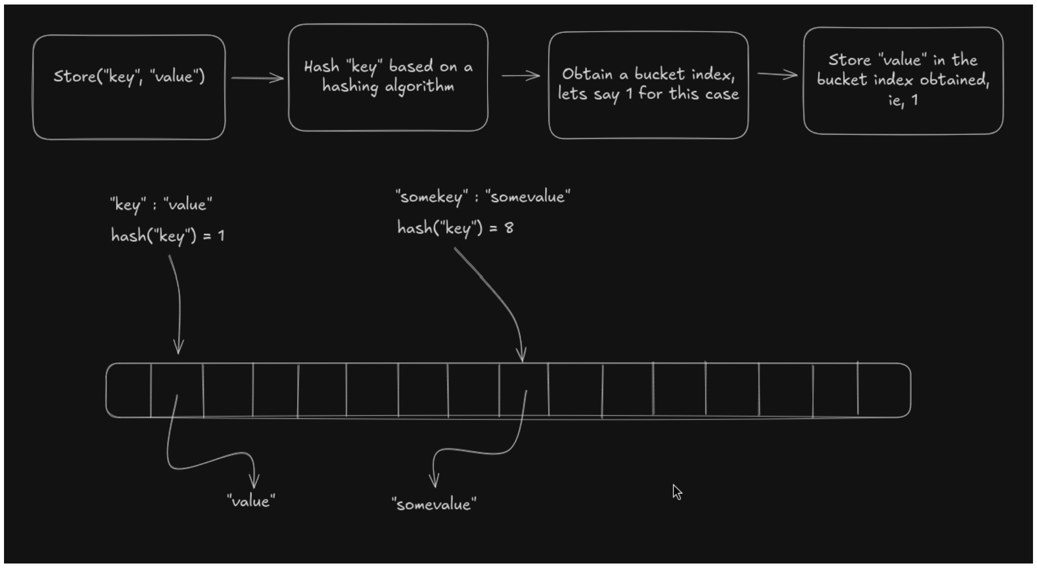

Store("key", "value")
Hash "key" based on a hashing algorithm
Obtain a bucket index, lets say 1 for this case
Store "value" in the bucket index obtained, ie, 1
"key" : "value"
hash("key") = 1
"somekey" : "somevalue"
hash("key") = 8
"value"
"somevalue"

Fig. 1. Hashmap structure

Internally, maps are implemented as an array of elements called buckets. The logic works by hashing the key of the given key-value pair to obtain a bucket index, which determines the position of the given value in the bucket. However, in rare cases, collisions occur when two different keys hash to the same bucket index. Various strategies exist to handle these collisions.

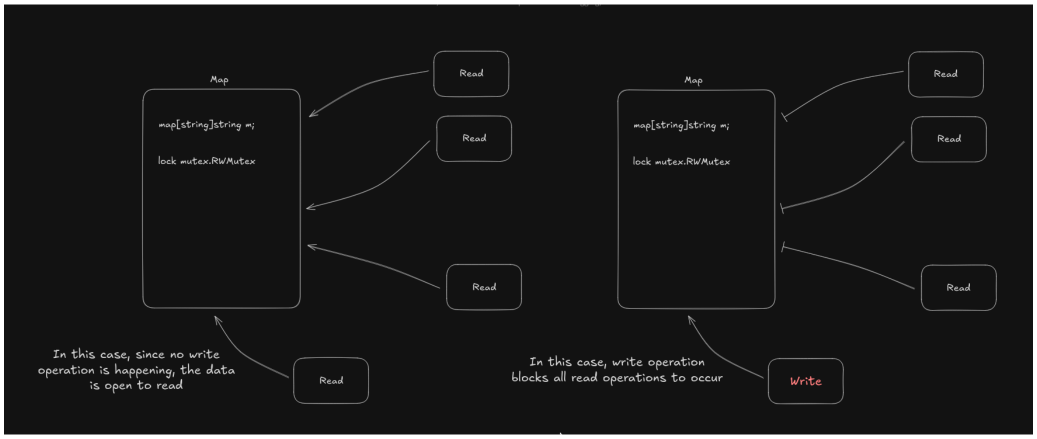
For example, In Separate Chaining, each bucket holds a linked list(or similar data structure). Where if collision occurs, a new node is appended. In Open Addressing, when collisions occur, the algorithm searches for the next available slot and places the value there.

2.2 Mutex-based Synchronization

Mutex, or lock-based synchronization, is one of the most widely used techniques for managing concurrent access. It works by creating and maintaining a lock on a data structure, such as a map. When a thread or goroutine acquires the lock for a read or write operation, any other thread attempting access must wait until the lock is released.

This can be visualized as a queue at a restaurant: each customer must wait their turn to place an order — analogous to threads waiting for access to the data. No one can skip ahead; all customers (i.e., threads) must wait until the current one has finished before proceeding.

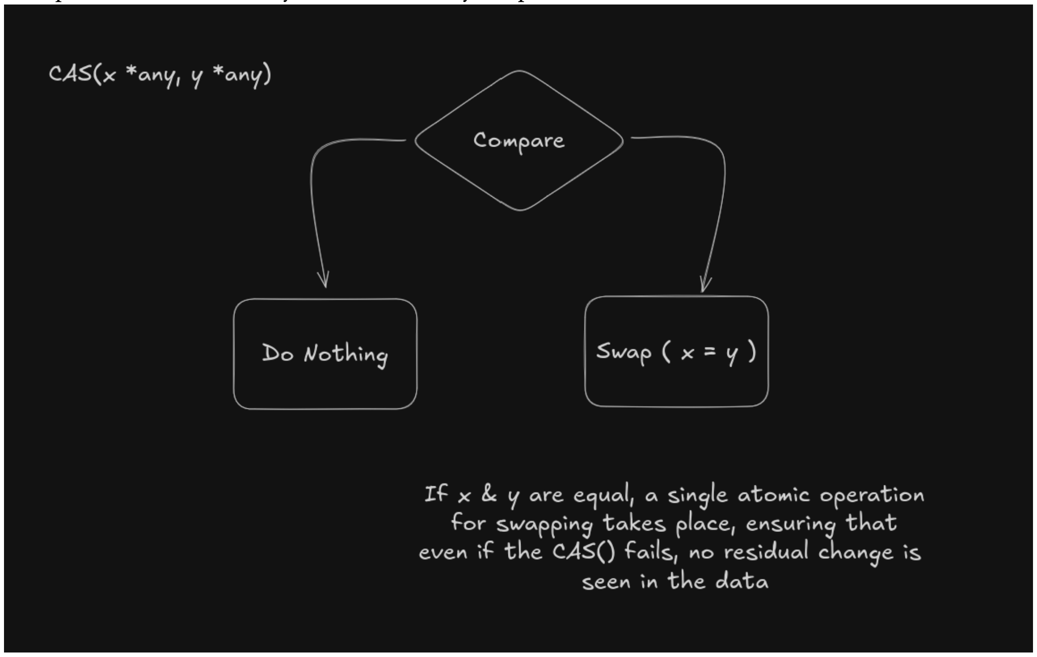
2.2.1 RWMutex. Unlike a Mutex, which creates a lock on the data structure for both read and write operations, RWMutex locks the data structure exclusively during *write* operations, allowing multiple threads to access and read the data while no write operation is being performed.


Map
map[string]string m;
lock mutex.RWMutex
Read
Read
Read
Read
In this case, since no write operation is happening, the data is open to read
Map
map[string]string m;
lock mutex.RWMutex
Read
Read
Read
Write
In this case, write operation blocks all read operations to occur

2.3 Lock-Free Synchronization

Lock-free synchronization relies on atomic operations to manage concurrent state changes without locks. Atomic operations run completely or not at all, with no partial effect on the given data. This means threads can actively access data without waiting.

If a race occurs, the data remains unaffected, and the thread retries the atomic operation until it succeeds. A common example is the hardware-supported Compare-And-Swap (CAS), which compares the current value at a memory address to an expected value and, if they match, atomically swaps it with a new value.

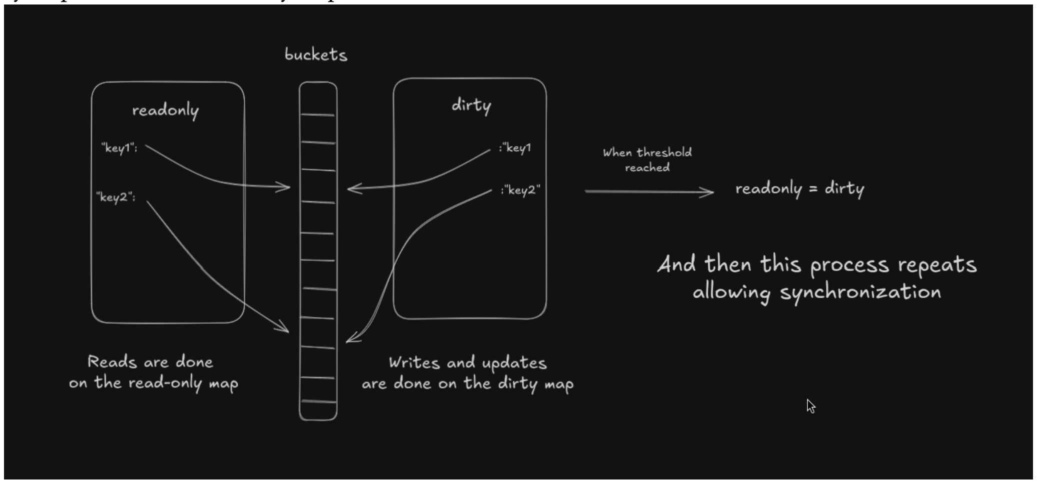

CAS(x *any, y *any)
Compare
Do Nothing
Swap (x = y)
If x & y are equal, a single atomic operation for swapping takes place, ensuring that even if the CAS() fails, no residual change is seen in the data

While atomicity is the foundation, lock-free algorithms differ in granularity. They typically operate on individual nodes or elements in a data structure, allowing higher concurrency. In contrast, applying atomicity at the level of the entire data structure limits concurrent modifications.

2.4 Partial Lock-Free Synchronization

Partial lock-free synchronization combines lock-free techniques for read operations with mutex-based locking for write operations. It combines the lock-free method for read operations and the Mutex method for write operations.

A partial lock-free implementation keeps two separate maps, one read-only lock-free map and one dirty Mutex-based map. All write operations are performed to the dirty map, and when the number of changes exceeds a set threshold, the dirty map is set as the read-only map.


buckets
readonly
"key1":
"key2":
dirty
:"key1
:"key2"
When threshold reached
readonly = dirty
And then this process repeats allowing synchronization
Reads are done on the read-only map
Writes and updates are done on the dirty map

Each of these maps holds a pointer to the same underlying element, ensuring that when an update occurs, the value is reflected in both maps rather than just one. Such a synchronization method is better aligned with high-read workloads.

3 Methodology

To accurately benchmark each of the concurrency control methods, all benchmarks are executed in a resource-limited, isolated, containerized environment to minimize external influence.

Each concurrency control method is tested under 3 operational conditions, which are,

- Write Heavy: 80% writes & 20% reads
- Read Heavy: 20% writes & 80% reads
- Balanced: 50% writes & 50% reads

During these operations, multiple performance metric are recorded.

3.0.1 Throughput. Here, throughput is defined as the total operations completed by the routines divided by the total time it took to complete those operations.

$$\text{Throughput} = \frac{\text{Total Operations Completed}}{\text{Total Time Taken}}$$

3.0.2 Latency. Here, Latency refers to the average time taken to complete an operation whether it be read or write operation.

$$\text{Latency} = \frac{\text{Total Time Taken by Operations}}{\text{Total Number of Operations}}$$

3.0.3 Memory Used. Here, Memory Used refers to the total amount of memory used by the program to complete all operations.

3.0.4 CPU Time. Here, CPU Time refers to the number of seconds the program utilized the CPU resources to execute.

```go
wg.Add(numRoutines)
for i := 0; i < numRoutines; i++ {
    go func(id int) {
        defer wg.Done()
        r := rand.New(rand.NewSource(time.Now().UnixNano() + int64(id)))
        read := 5
        if OpType == benchmark.ReadHeavy {
            read = 8
        } else if OpType == benchmark.WriteHeavy {
            read = 2
        }

        for j := 0; j < opsPerRoutine; j++ {
            key := fmt.Sprintf("r%d_op%d", id, j)
            if r.Intn(10) < read {
                m.Load(key)
            } else {
                m.Store(key, j)
            }
        }
    }(i)
}
wg.Wait()
```

In the above code snippet, we create a wait-group, with the number of operations we will perform (`numRoutines`). Making the code wait till all the routines are finished (`wg.Wait()`).

For each routine, we define the number of operations it will perform (`opsPerRoutine`) and also the type of operation (`opType`) which can be Read Heavy, Write Heavy or Balanced. And then we run the operation on the given map implementation based on the percent chance of each operation (Read or Write) to occur.

```go
func Run() Results {
        return Results{
                LockFree:        RunLockfree(50),
                RWMutex:         RunRWMutex(50),
                PartialLockfree: RunPartialLockfree(50),
        }
}

func RunLockfree(n int) []Result {
	items := make([]Result, 0, n)
	for i := 0; i < n; i++ {
		m := skipmap.New[string, int]()
		r := Result{
			Balanced:   mapimpl.Simulate(m, 1000, 10000, benchmark.Balanced,
			↪  benchmark.LockFree),
			ReadHeavy:  mapimpl.Simulate(m, 1000, 10000, benchmark.ReadHeavy,
			↪  benchmark.LockFree),
			WriteHeavy: mapimpl.Simulate(m, 1000, 10000, benchmark.WriteHeavy,
			↪  benchmark.LockFree),
		}
		items = append(items, r)
	}

	return items
}
```

As you could see in this snippet, each synchronization method is run 50 times to achieve an unbiased mean value for evaluation. For each simulation, 1,000 routines concurrently perform 10,000 operations.

3.1 RWMutex Implementation

```go
type RWMutexMap struct {
	Map map[string]int
	mu  sync.RWMutex
}

func NewRWMutex() *RWMutexMap {
	return &RWMutexMap{
		Map: make(map[string]int),
		mu:  sync.RWMutex{},
	}
}

func (m *RWMutexMap) Store(key string, val int) {
	m.mu.Lock()
	defer m.mu.Unlock()
	m.Map[key] = val
}

func (m *RWMutexMap) Load(key string) (int, bool) {
	m.mu.RLock()
	defer m.mu.RUnlock()
```

```
        v, ok := m.Map[key]
        return v, ok
}
```

3.2 Lock-free Implementation

The `"github.com/zhangyunhao116/skipmap"` package provides the `skipmap` data structure which uses a lock-free implementation.

3.3 Partial Lock-free Implementation

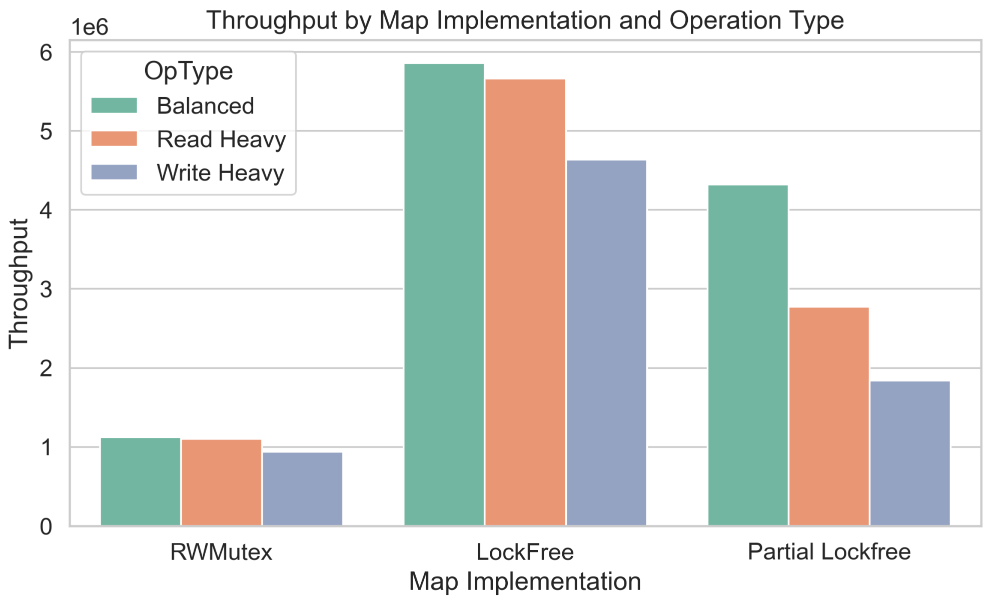
```
type PartialLockfree struct {
        m *sync.Map
}

func NewPartialLockfree() *PartialLockfree {
        return &PartialLockfree{m: &sync.Map{}}
}

func (s *PartialLockfree) Load(key string) (int, bool) {
        val, ok := s.m.Load(key)
        if !ok {
                return 0, false
        }
        intVal, ok := val.(int)
        return intVal, ok
}

func (s *PartialLockfree) Store(key string, value int) {
        s.m.Store(key, value)
}
```

The standard `sync` package provides the `sync.Map{}` data structure which uses a partial lock-free implementation.

4 Results & Discussions

The benchmarks were performed 50 times, and the average values were taken for the graphs. For each implementation — Lock-Free, RWMutex, and Partial Lock-Free — and each workload type — Read-Heavy, Write-Heavy, and Balanced — the simulation was run with 1000 routines performing 10,000 operations each. Each graph shows the results for each implementation and its respective workload type.

4.1 Throughput



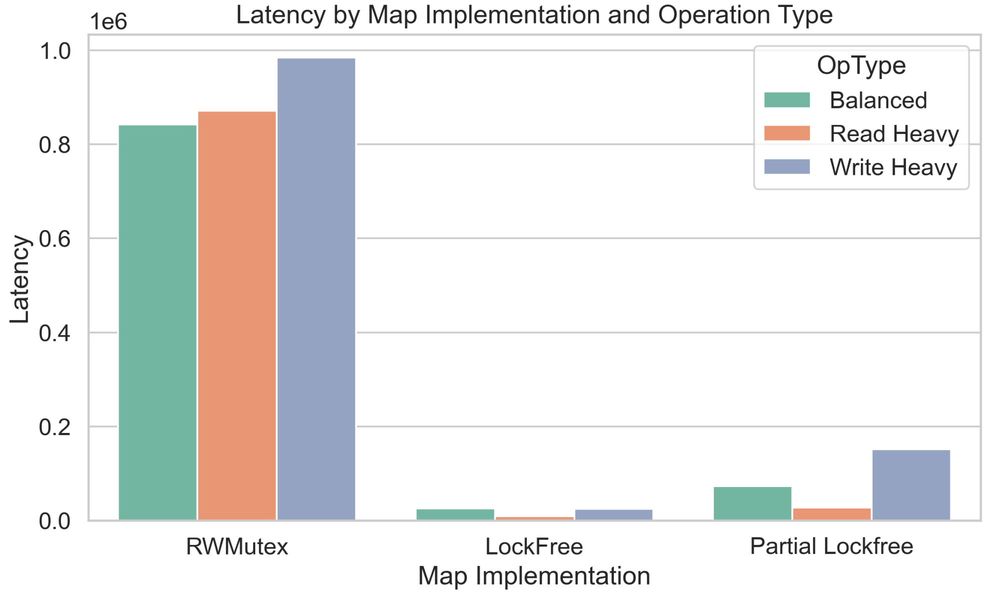
From the resulting graph, we can conclude that the Lock-Free implementation outperforms RWMutex and Partial Lock-Free. This is expected, as Lock-Free does not use any locks and is a wait-free algorithm. RWMutex performs the worst due to its heavy reliance on locks, whereas Partial Lock-Free uses them only for writes.

For all implementations, a significant decrease in throughput is observed during write-heavy workloads. This aligns with Lock-Free and Partial Lock-Free being optimized for read-heavy loads. This is also why, for RWMutex, the relative decrease in throughput for writes is smaller — since it is designed to be more balanced.

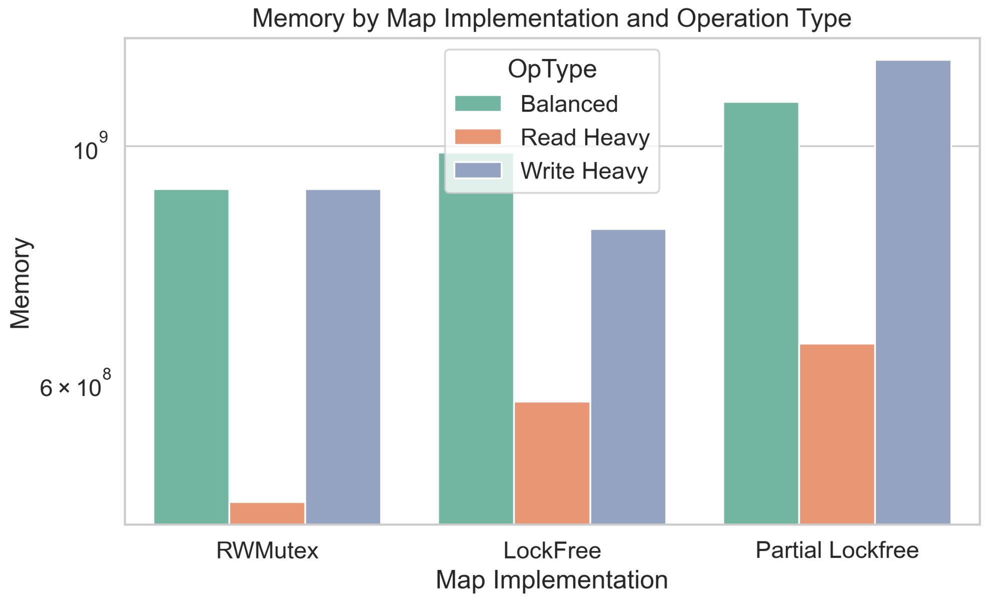
4.2 Latency

The Lock-Free implementation consistently shows the lowest latencies across all workloads, whereas RWMutex performs the worst due to its high reliance on locks. We can also observe that, for Lock-Free and Partial Lock-Free, the read-heavy latencies are significantly lower, which aligns with their read-heavy optimizations.

All implementations experience an increase in latency for write-heavy loads, which is expected since a write operation is inherently more complex than a read operation. This is most evident in Partial Lock-Free and RWMutex, which use locks for write operations and therefore experience the largest increase in latency.



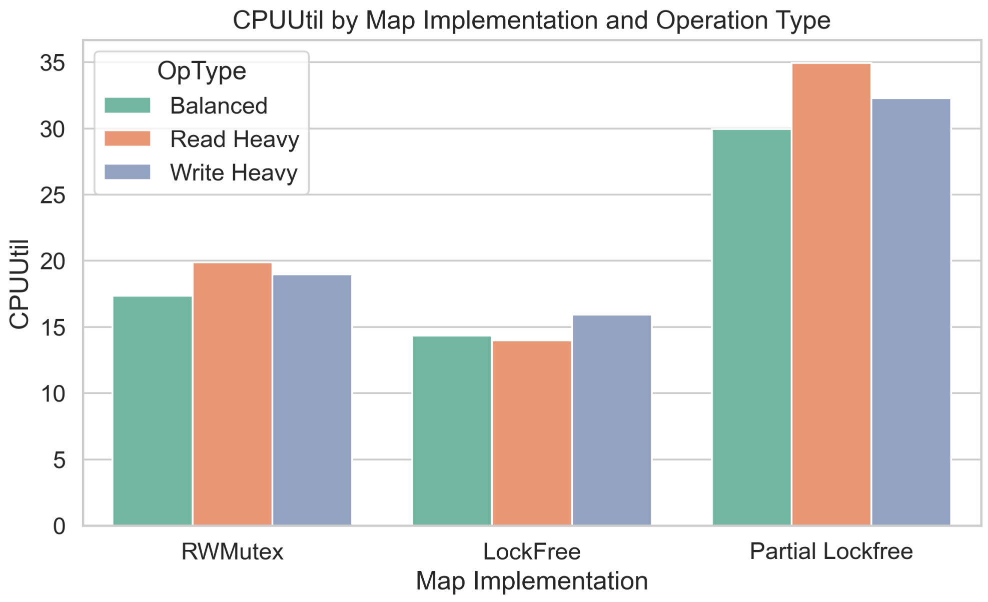
4.3 Memory



We observe that all implementations consume a substantial amount of memory, but Partial Lock-Free tends to use significantly more. This is consistent with the fact that Partial Lock-Free maintains two maps instead of one.

RWMutex and Lock-Free consume almost similar amounts of memory, but for read-heavy operations, RWMutex uses significantly less compared to other operations. This is due to its use of locks, which have less overhead compared to the other implementations.

4.4 CPU Utilization



We observe that Partial Lock-Free consumes significantly more CPU time than the other implementations. This is due to its internal promotion process and the CPU-intensive nature of its lock usage.

RWMutex uses less CPU during write-heavy workloads because blocked threads tend to sleep, whereas Lock-Free uses slightly more CPU due to loops and retries caused by CAS or other atomic operations.

5 Conclusion

Lock-Free showed the highest throughput and lowest latency, especially in read-heavy and balanced workloads, due to its wait-free design. RWMutex performed best under write-heavy loads, offering stability but at the cost of blocking. Partial Lock-Free delivered middle-ground results, with additional CPU and memory overhead from its dual-map design.

Although the Lock-Free implementation shows better performance in most cases, it is considered less safe compared to RWMutex or other mutex-based approaches. This is because Lock-Free algorithms have increased complexity in handling edge cases and are thus more prone to errors.

Future work could focus on making Lock-Free safer through improved memory management and verification tools. Adaptive strategies that switch methods based on workload could also combine performance with reliability.

References

[1] The Go Authors. *The Go Programming Language: sync Package Documentation.* Available: https://pkg.go.dev/sync
[2] Zhang, Y. *skipmap: A Lock-Free Concurrent Ordered Map for Go.* GitHub repository: https://github.com/zhangyunhao116/skipmap
[3] Herlihy, M., & Shavit, N. (2011). *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers.

[4] Michael, M. M., & Scott, M. L. (1996). *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC).*

[5] Dice, D., Shalev, O., & Shavit, N. (2006). *Transactional Locking II. Proceedings of the 20th International Symposium on Distributed Computing (DISC).*

[6] Preshing, J. *Understanding the sync.Map Internals in Go.* VictoriaMetrics Engineering Blog. Available: https://victoriametrics.com/blog/go-sync-map/

[7] Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* O'Reilly Media. [Concurrency control and consistency concepts]

[8] Bryant, R. E., & O'Hallaron, D. R. (2015). *Computer Systems: A Programmer's Perspective* (3rd ed.). Pearson. [Chapters on concurrency and synchronization in operating systems]

References