SyncMap/LockFree vs Mutex

That various members of the Go team discovered less than ideal performance when a `sync.RWMutex` was utilized in systems that were deployed on many, many cores. And by many I'm not really talking about your typical 8 or 16 core setup but primarily those server setups that are well beyond that. Where the number of cores really starts to show in the form of highly contentious code such as when the `sync.RWMutex` is used.

And now we've arrived as to why the `sync.Map` was created. The Go team identified situations in the standard lib where performance wasn't great. There were cases where items were fetched from data structures wrapped in a `sync.RWMutex`, under high read scenarios while deployed on very high multi-core setups and performance suffered considerably.

Go's Runtime Scheduler Model

1. **M:N Scheduling Model**
    a. Go uses an M:N scheduler where M goroutines are multiplexed onto N OS threads

   b. The OS threads are then scheduled onto P physical processors (cores) by the operating system
   c. This three-level hierarchy is often described as the GPM model: Goroutines, Processors (P), and Machine threads (M)
2. **Components of the Scheduler**
   a. **G (Goroutine)**: A goroutine is the unit of execution in Go
   b. **M (Machine)**: An OS thread that can execute goroutines
   c. **P (Processor)**: A logical processor that acts as a resource context for running goroutines (roughly corresponds to a CPU core)
3. **Default Configuration**
   a. By default, Go creates as many P as there are CPU cores (runtime.GOMAXPROCS)
   b. You can adjust GOMAXPROCS to control the maximum number of cores your program can use

Cache Coherence Protocol Overhead

When multiple CPU cores share memory, they maintain local caches of that memory for performance. However, this creates a coherence problem - if one core modifies data, other cores need to know about it. Modern CPUs implement cache coherence protocols to handle this automatically.

When a lock is acquired and released:

- The CPU must perform expensive memory barriers/fences
- Cache lines containing the lock must be transferred between cores
- This creates traffic on the inter-core communication fabric (like Intel's QuickPath Interconnect or AMD's Infinity Fabric)
- **Cache Coherence Overhead**:
- Modern CPUs use **cache coherence protocols** (e.g., MESI) to keep caches in sync across cores.
- When one core modifies a shared variable, it invalidates copies in other cores' caches, leading to high **cache misses** and **memory stalls**.
- A frequently updated lock (e.g., in `sync.RWMutex`) forces cores to constantly update their caches, leading to performance degradation.
- As core count increases, this traffic becomes a significant bottleneck.

Lock Contention Scaling Issues

With traditional locks like `sync.RWMutex`:

- Even though multiple readers can hold the lock simultaneously, they all contend when acquiring/releasing it
- Each reader must update shared state, forcing all cores to synchronize
- At high core counts (32, 64, 128+ cores), the synchronization overhead becomes overwhelming
- Performance often degrades exponentially rather than linearly with increased contention