Optimistic Concurrency Control for Real-world Go Programs

Zhizhou (Chris) Zhang[1], Milind Chabbi[2], Adam Welc[2], and Timothy Sherwood[1]

[1] University of California, Santa Barbara

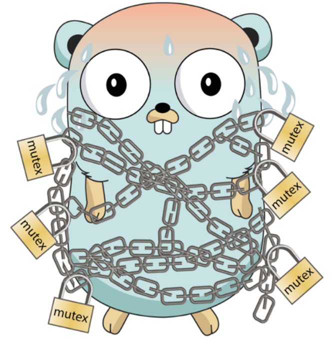
[2] Programming System Group, Uber Technologies

Golang and Concurrency

- Go is a modern programming language
  - Has similarities with C
- Go is gaining popularity
- Go has concurrency at its core
  - Prefix “**go**” to any function call for concurrent execution
  - Concurrent function (goroutine): light-weight thread
- Go has 2 concurrency control models:
  - Shared memory (**Mutex** lock)
  - Message passing (Channel)

Mutex Lock → Pessimistic Concurrency Control

- Locks preemptively serialize goroutines
  - Regardless of shared variable
- Lock-based serialization can be expensive:
  - 5~30% locking overhead in production code




```
func update() {
    m.Lock()
    count++
    m.Unlock()
}
```

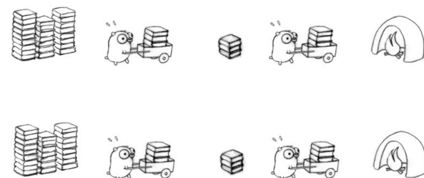
Do we need to be pessimistic when conflicts are rare?

Optimism with Transactional Memory

- Speculatively execute concurrent regions assuming no conflicting shared memory accesses
- Abort and retry in case of memory conflict (at least one access is a write)

Well-suited for boosting performance of otherwise lock-protected critical sections (a.k.a. transactional lock elision)

Hardware Transactional Memory (HTM)



- Architectural support for transactional memory
- Available in Intel, PowerPC, and Arm…
- Exploits cache coherence protocol to detect conflict in programmer specified code regions
- Executes transactions **in parallel** speculatively
- Many previous studies show HTM is capable of lock elision

```
func update_HTM() {
     tx.begin()
     count++
     tx.end()
}
```

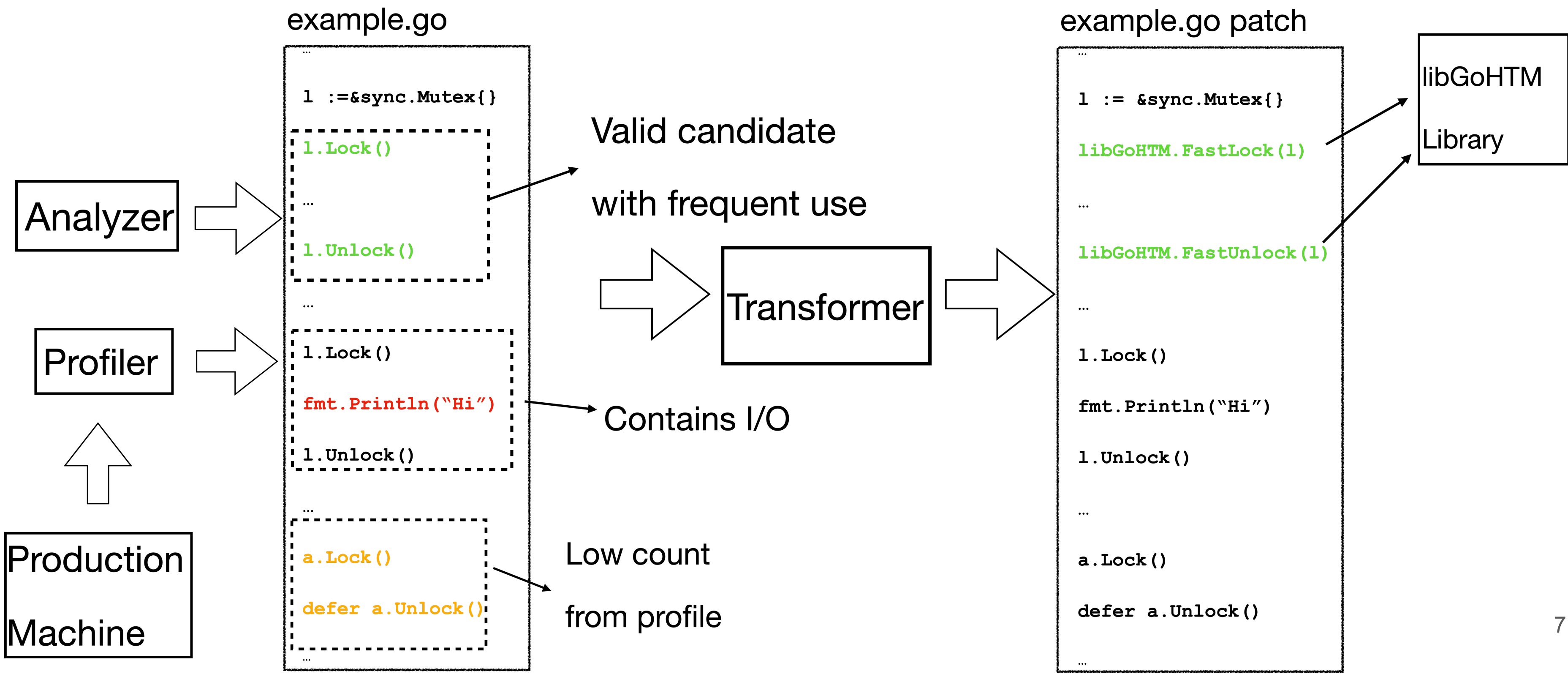
Why not replace all locks in Go with HTM?

Challenges

- Limitations of HTM implementations
  - Limited capacity in a transactional region
  - Unfriendly instructions to HTM, e.g. syscall
- Therefore performance gain is **NOT** guaranteed
  - Performance can degrade
- Opportunity:
  - Average Golang developers are not concurrency experts
  - Many low-hanging fruits to optimize critical sections in Go programs

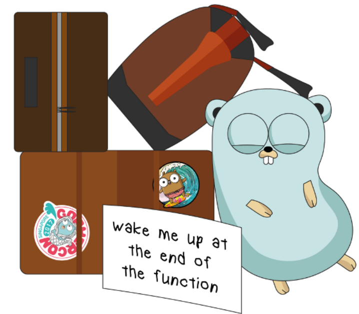
Our Solution: GOCC

- A framework to automatically generate HTM patch for Go code



Implementation Challenges of GOCC

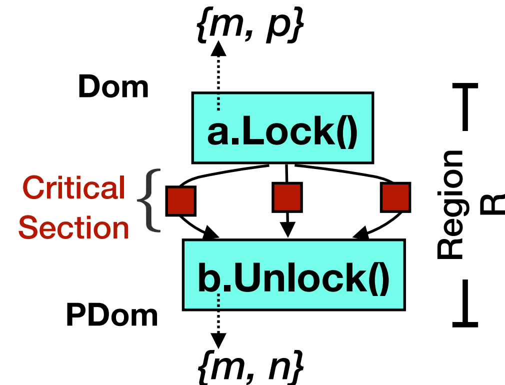
- Automatically identifying lock-protected critical sections accurately and efficiently
- Golang quirks:
  - defer
- Function calls in critical sections
- Nested locks




```
func update_defer() {
    m.Lock()
    defer m.Unlock()
    if counter == nil {
        return 0
    }
    return counter.Get()
}
```

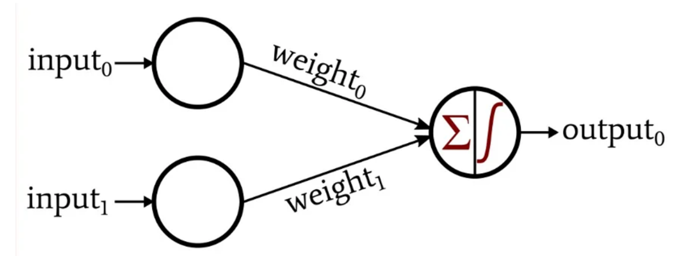
GOCC Lock Identification

- Uses dominator-based analysis to identify lock/unlock pair
- Analyzes instructions protected by lock/unlock pair
    - Eliminates lock/unlock pairs containing HTM unfriendly instructions
- Guarantees the correctness of the transformation
- More details please refer to the paper



GOCC Runtime Contention Management

- Uses historic HTM success/failure of a critical section to guide future decisions
- Employs a light-weight hash-based perceptron design
- Incorporate several techniques from previous HTM studies to avoid performance collapse

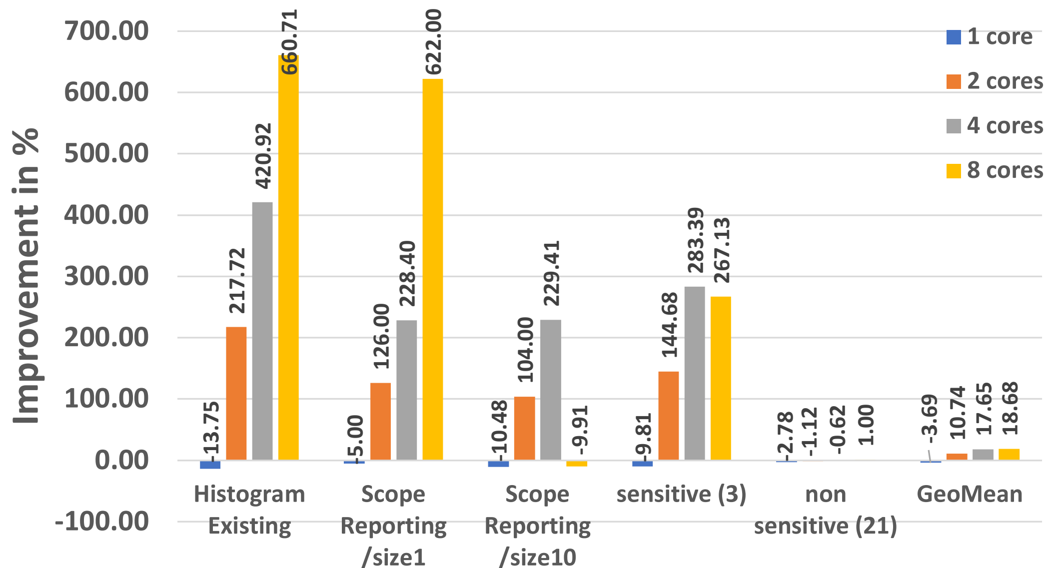


Evaluation

- Test on Intel i9-9900KF with RTM support, 8 cores/16 threads with SMT
- 32GB memory
- Golang version is 1.15.2
- Go lacks standard benchmark programs
- We choose Tally, which is widely used in industry
  - Several other benchmarks based on their popularity from GitHub, i.e. concurrent set

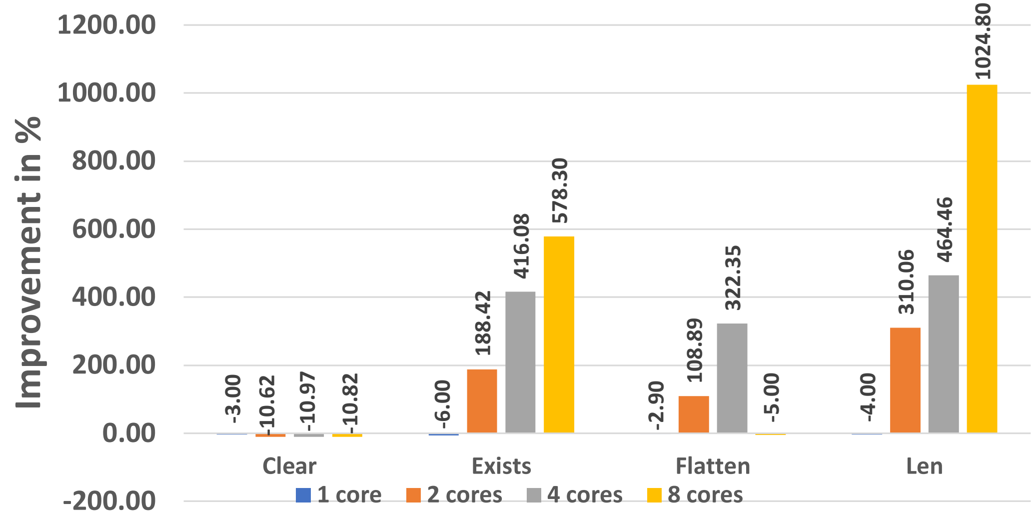
Benchmark 1: Tally

- Significant (6x) speedup
- Scalable
- Negligible slowdown



Benchmark 2: Concurrent set

- Significant (10x) speedup
- Scalable
- Negligible slowdown



Conclusions

Thank you!
Any questions?

- Go is concurrency-first language
  - Suffers often due to lock-based serialization
- GOCC: speedup Go concurrent programs using HTM in place of lock
- Key idea: change **some** of the lock **some** time
- Generates scalable performance with minimum slowdown
- Source-to-source transformer at: https://github.com/uber-research/GOCC

Questions: zhizhouzhang@ucsb.edu