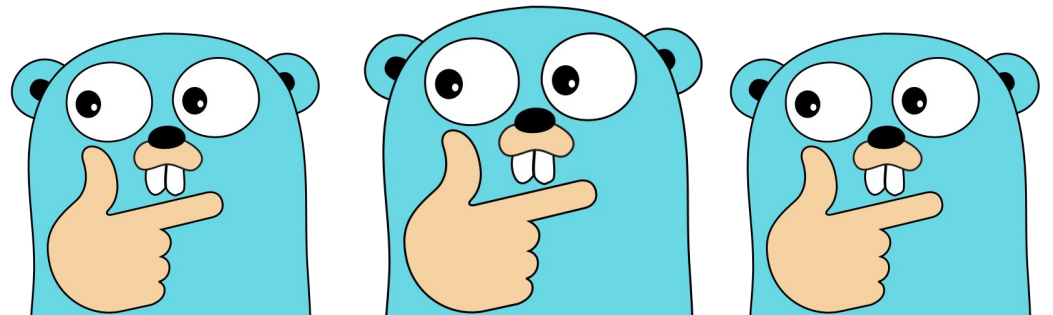


Is it time to re-sync?

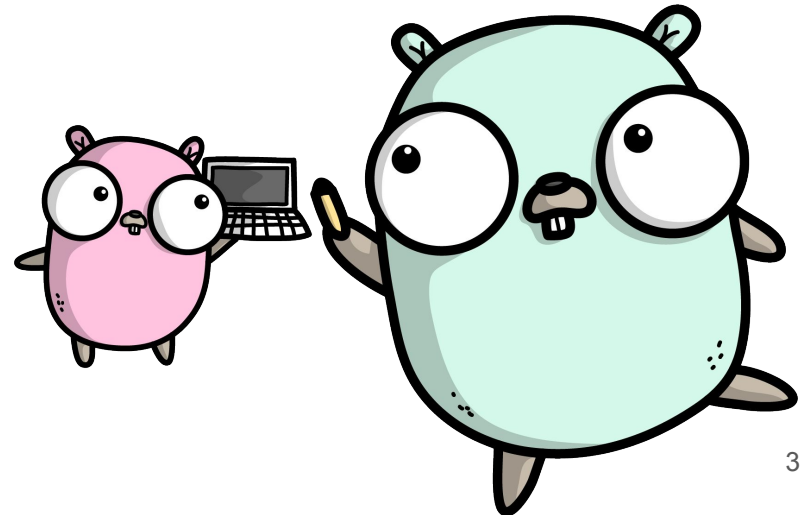
Андрей Печкуров



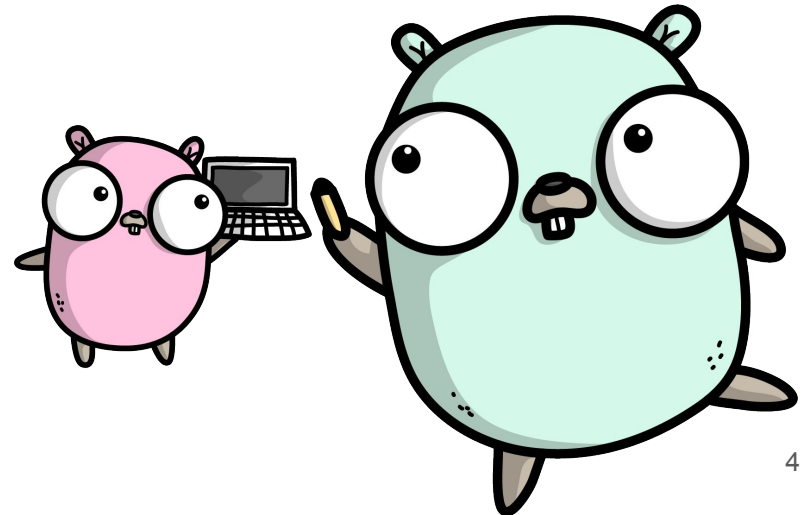
re-sync?

Поговорим про
пакеты sync, sync/atomic, каналы
И ВОТ ЭТО ВОТ ВСЕ.

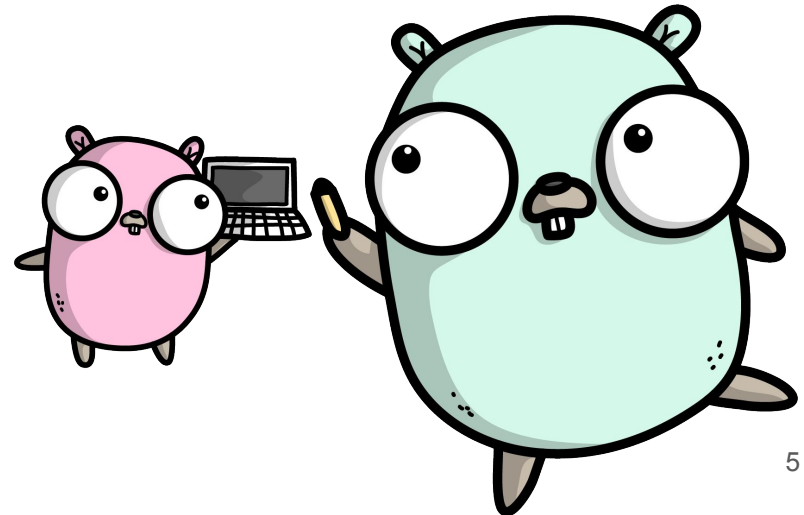
Можно ли улучшить масштабируемость
`sync.RWMutex`?



Возможно ли обогнать каналы в
производительности?



Верно ли то, что нет ничего лучше
map+RWMutex, ну, или sync.Map?



О докладчике

Обожаю open source. Node.js core contributor

Интересы:

веб, архитектура, распределенные системы,
производительность

Можно найти тут:

<https://twitter.com/AndreyPechkurov>

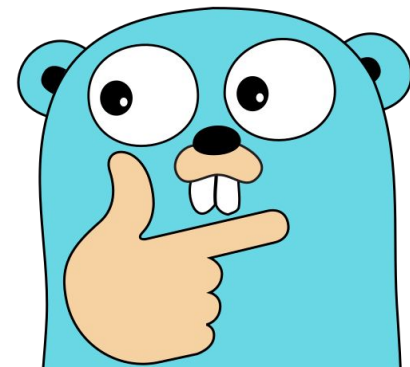
<https://github.com/puzpuzpuz>

<https://medium.com/@apechkurov>

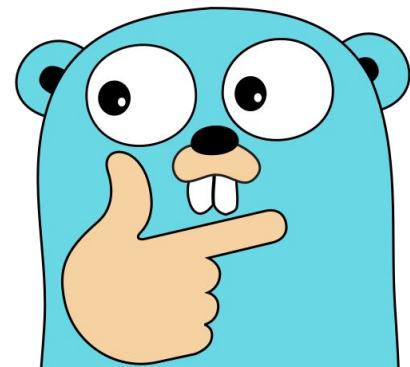


План на сегодня

1. Железо, софт и законы вселенной в разрезе многопоточности
2. Изобретение альтернатив встроенным структурам данных



Железо, софт и законы вселенной в разрезе многопоточности

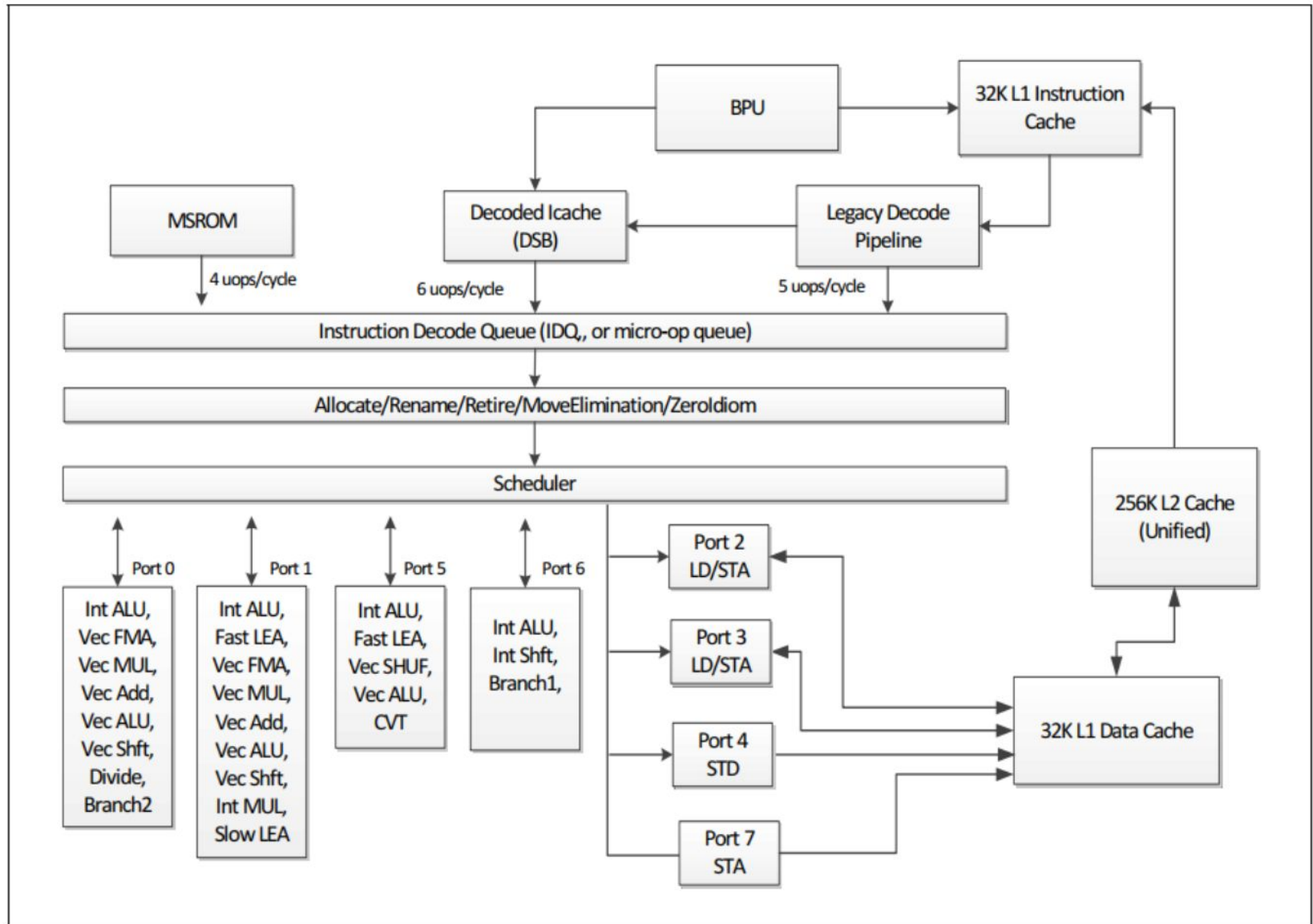


Mechanical sympathy

“You don’t have to be an engineer to be a racing driver, but you do have to have Mechanical Sympathy.” – Jackie Stewart, автогонщик

Martin Thompson применил этот принцип к софту.

CPU Microarchitecture (Intel Skylake)



Конвейер

Instruction	Clock cycle								
	1	2	3	4	5	6	7	8	9
Instruction x	IF	ID	EXE	MEM	WB				
Instruction x+1		IF	ID	EXE	MEM	WB			
Instruction x+2			IF	ID	EXE	MEM	WB		
Instruction x+3				IF	ID	EXE	MEM	WB	
Instruction x+4					IF	ID	EXE	MEM	WB

Пример конвейера с 5ю стадиями:

1. Instruction fetch (IF)
2. Instruction decode (ID)
3. Execute (EXE)
4. Memory access (MEM)
5. Write back (WB)

Конвейер

$$\text{Time per instruction on pipelined machine} = \frac{\text{Time per instruction on non-pipelined machine}}{\text{Number of pipe stages}}$$

Суперскалярность

Instruction	Clock cycle					
	1	2	3	4	5	6
Instruction x	IF	ID	EXE	MEM	WB	
Instruction x+1	IF	ID	EXE	MEM	WB	
Instruction x+2		IF	ID	EXE	MEM	WB
Instruction x+3		IF	ID	EXE	MEM	WB

Пример двухрядного конвейера

Внеочередное выполнение

Instruction	Clock cycle									
	1	2	3	4	5	6	7	8	9	10
Instruction x	IF	ID	EXE	MEM	WB					
Instruction x+1		IF	ID			EXE	MEM	WB		
Instruction x+2			IF	ID	EXE	MEM			WB	
Instruction x+3				IF	ID		EXE	MEM		WB

Спекуляция

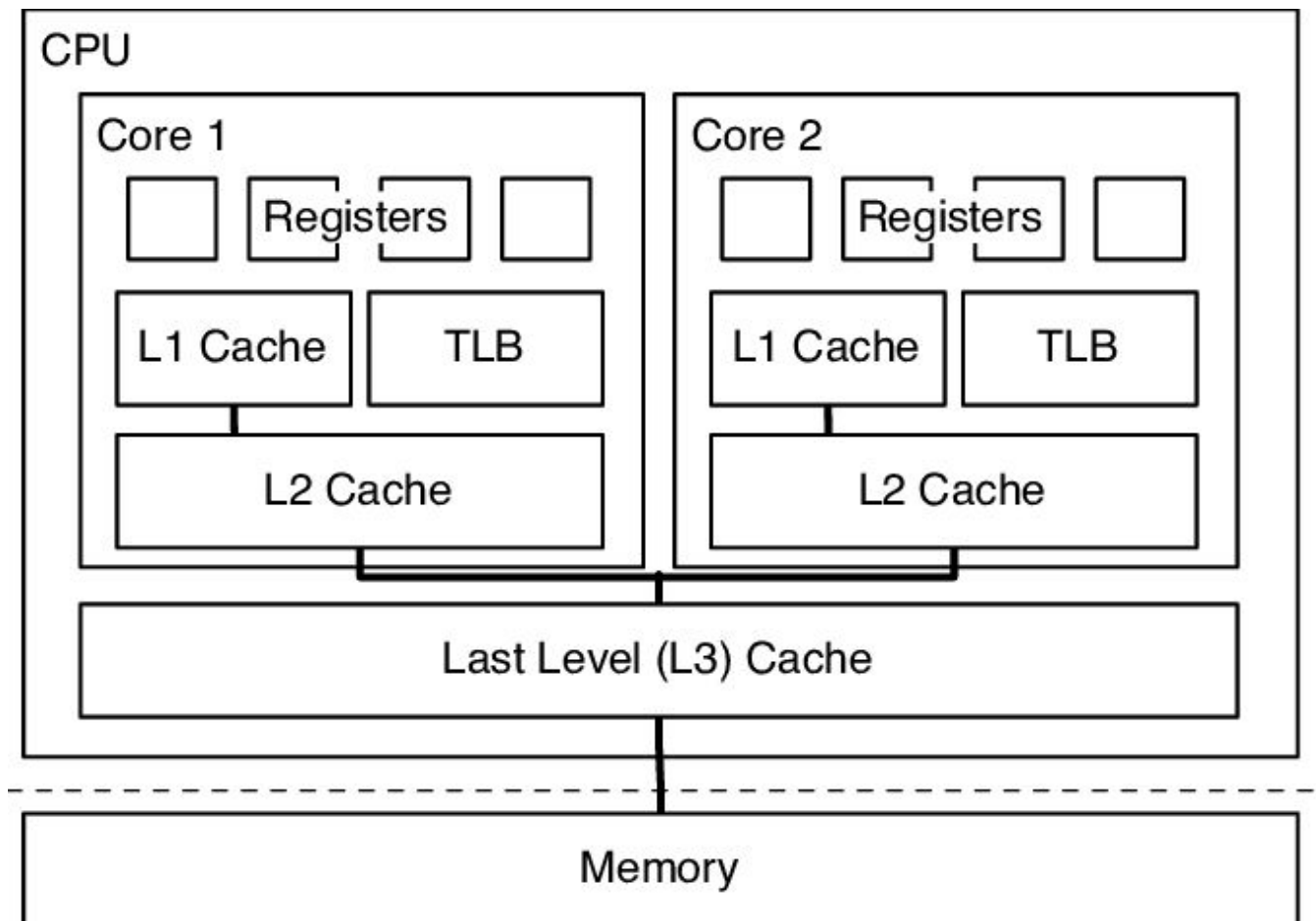
Без спекуляции

Instruction	Clock cycle							
	1	2	3	4	5	6	7	8
BRANCH (a < b)	IF	ID	EXE	MEM	WB			
CALL foo				IF	ID	EXE	MEM	WB
// INSTR from foo					IF	ID	EXE	MEM

Со спекуляцией

Instruction	Clock cycle							
	1	2	3	4	5	6	7	8
BRANCH (a < b)	IF	ID	EXE	MEM	WB			
CALL foo		IF*	ID*	EXE	MEM	WB		
// INSTR from foo			IF*	ID	EXE	MEM	WB	

Кэши ЦП (Intel Nehalem)



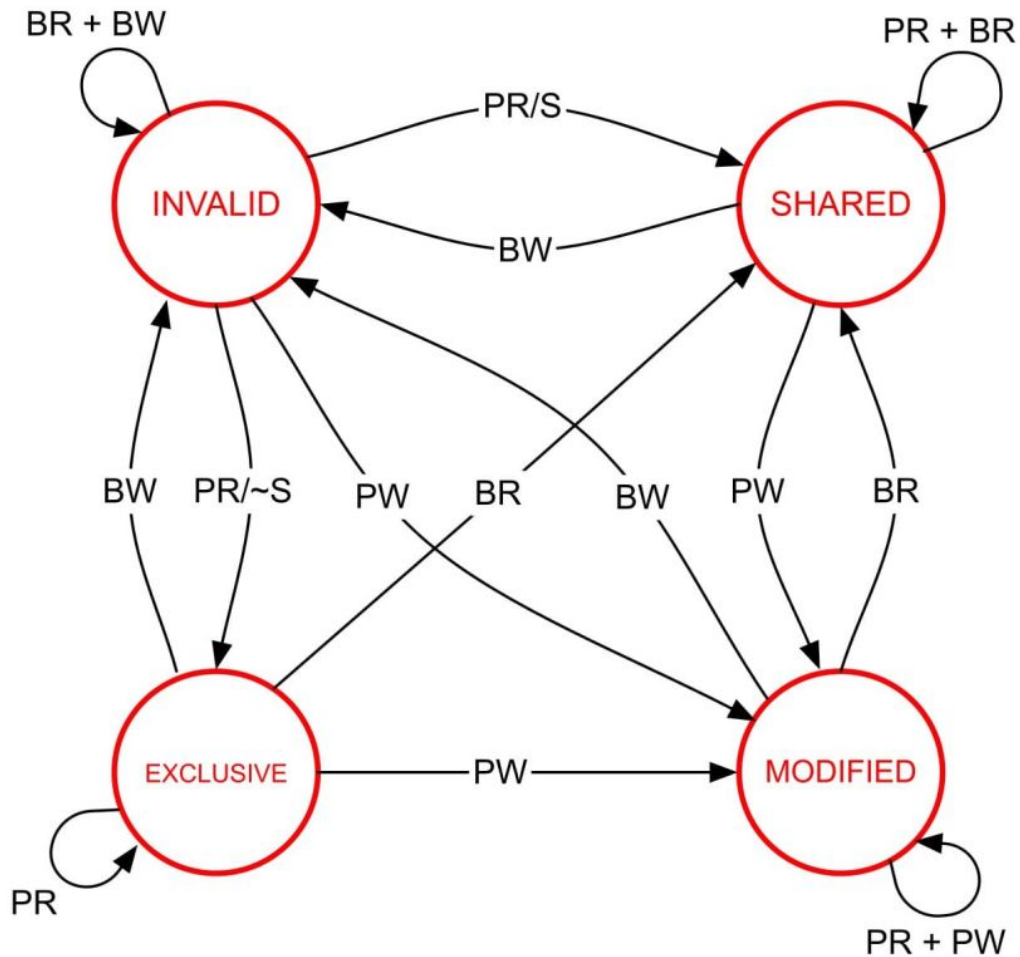
L1 cache - D-cache + I-cache

TLB - translation lookaside buffer

Кэши ЦП (Intel i7-6700, Skylake)

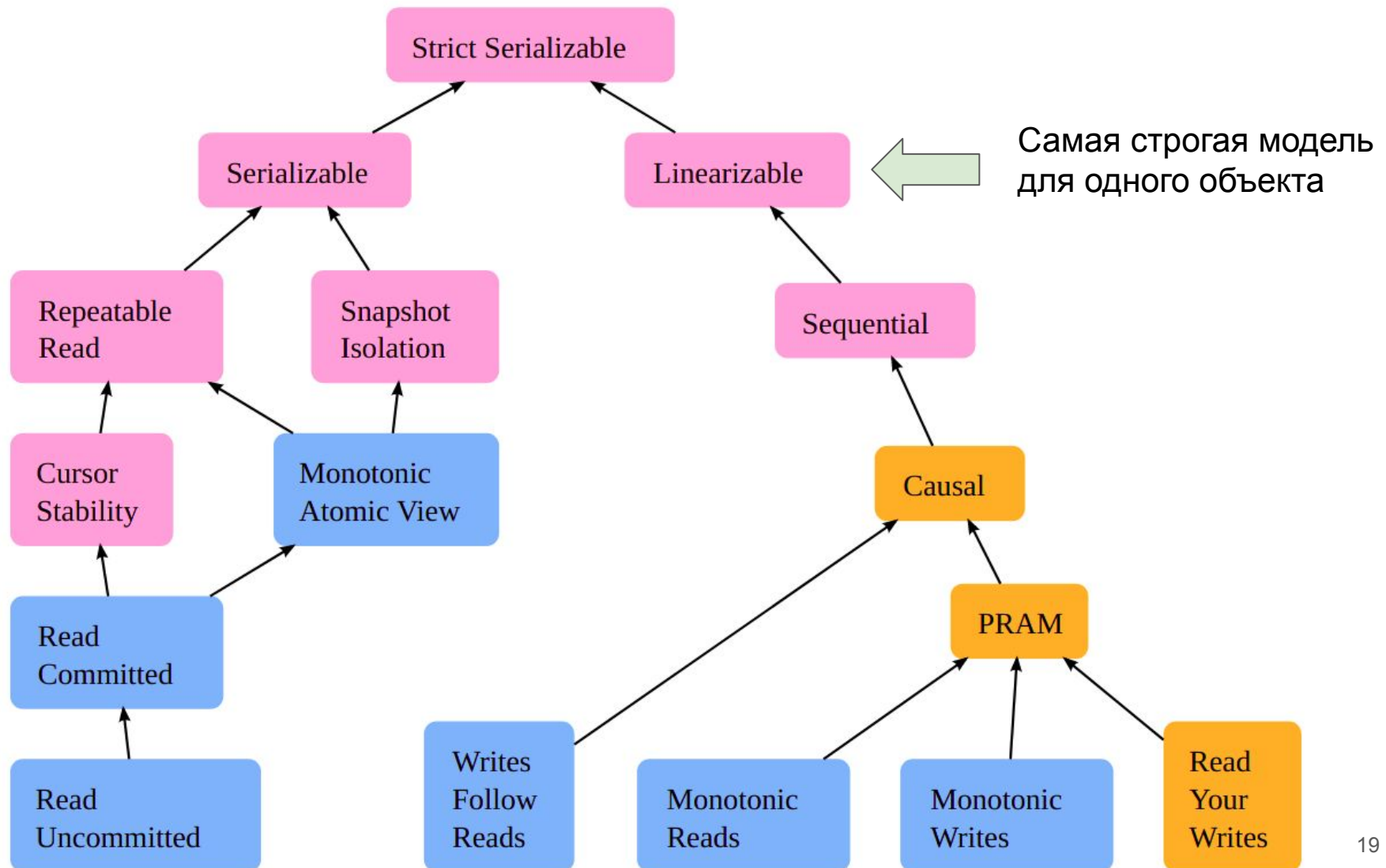
Cache level	Latency	Description
L1 cache	4-5 cycles	D-cache: 32 KB, 64 B/line, 8-WAY I-cache: 32 KB, 64 B/line, 8-WAY
L2 cache	12 cycles	256 KB, 64 B/line, 4-WAY
L3 cache	40-50 cycles	8 MB, 64 B/line, 16-WAY
Memory	42 cycles + 51 ns	

Согласованность кэша (e.g. MESI, MESIF, MOESI)



PR = processor read BR = observed bus read
PW = processor write BW = observed bus write
S/~S = shared/NOTshared

Модели согласованности



Аппаратные модели памяти

Атомарность, видимость и порядок

- x86 - Total Store Order (TSO)
- ARM - более слабая модель; ARMv8 - multicopy atomic

// Thread 1	// Thread 2
x = 1	r1 = y
y = 1	r2 = x

Может ли поток 2 наблюдать $r1 = 1$, $r2 = 0$?

Виды неблокирующих алгоритмов

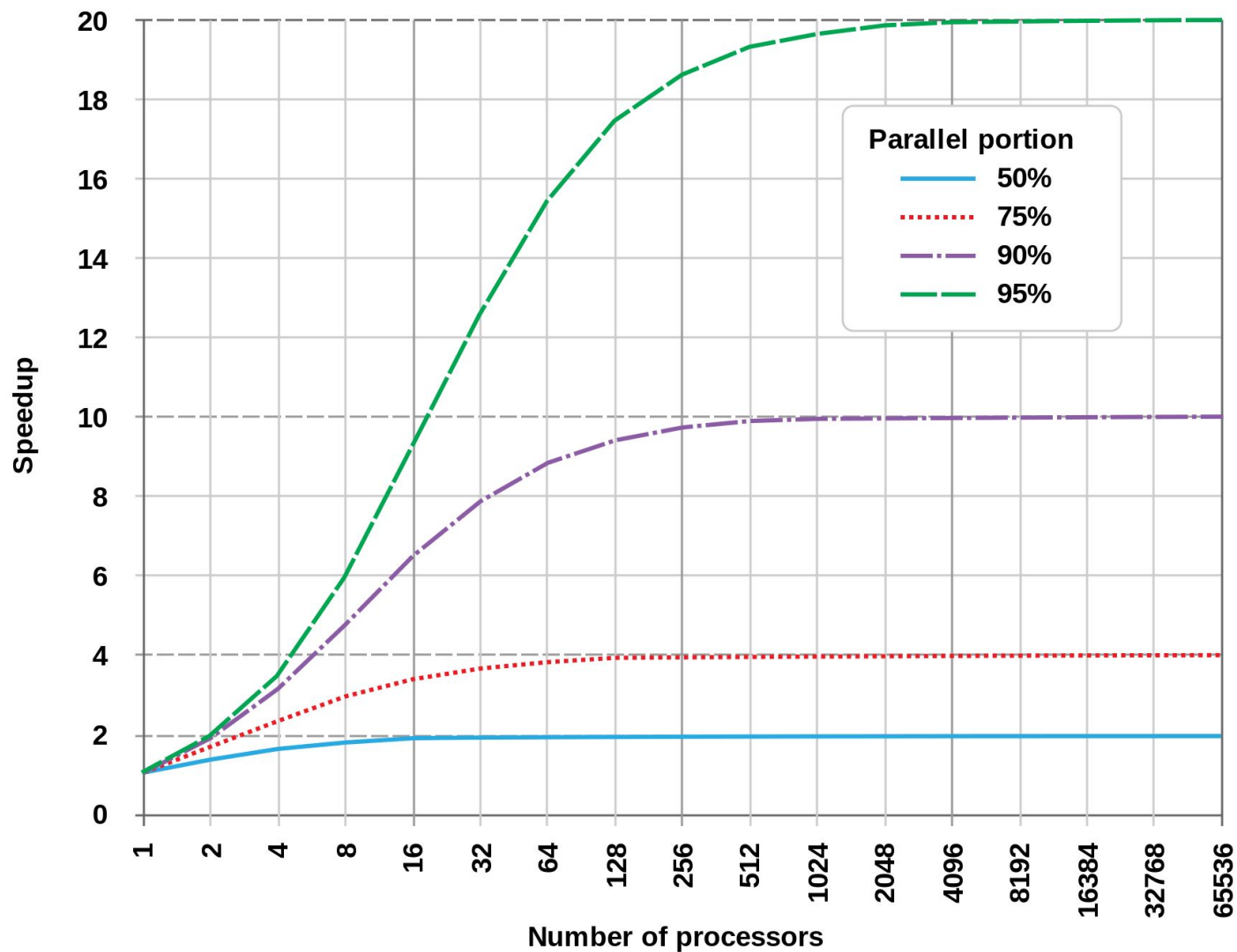
- Wait-free - гарантированный прогресс за конечное число шагов для каждого потока/процесса
- Lock-free - гарантированный прогресс приложения/системы, т.е. хотя бы одного потока/процесса
- Obstruction-free - прогресс каждого потока/процесса возможен в отсутствии препятствий со стороны конкурентов (contention)

Закон Амдала

$$S_p = \frac{1}{\alpha + \frac{1 - \alpha}{p}}$$

α - доля последовательных расчетов
 p - число узлов (например, ядер)

Закон Амдала



Universal Scalability Law

$$C(p) = \frac{p}{1 + \sigma(p - 1) + \kappa p(p - 1)}$$

p - число узлов (например, ядер)

σ (α в законе Амдала) - доля последовательных расчетов (contention)

κ - "цена" коммуникации между узлами

Director's Cut

- HW prefetcher
- NUMA (non-uniform memory access)
- SIMD (single instruction, multiple data)
- SMT (simultaneous multithreading)
- Много чего еще

Что сказал бы Халк?



Правило Халка №1

"Нет записи в общую память - чтения идти хорошо"*

*Перевод:

"идти хорошо"

значит

"масштабироваться линейно"

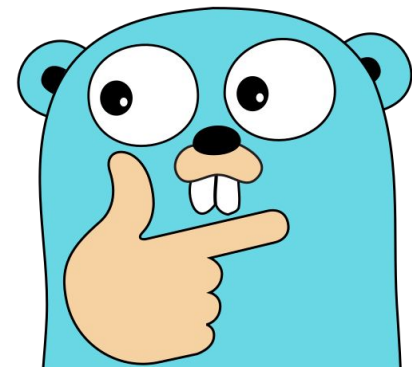


Правило Халка №2

"Редкие записи в общую память - записи идти неплохо"



Изобретение альтернатив встроенным структурам данных



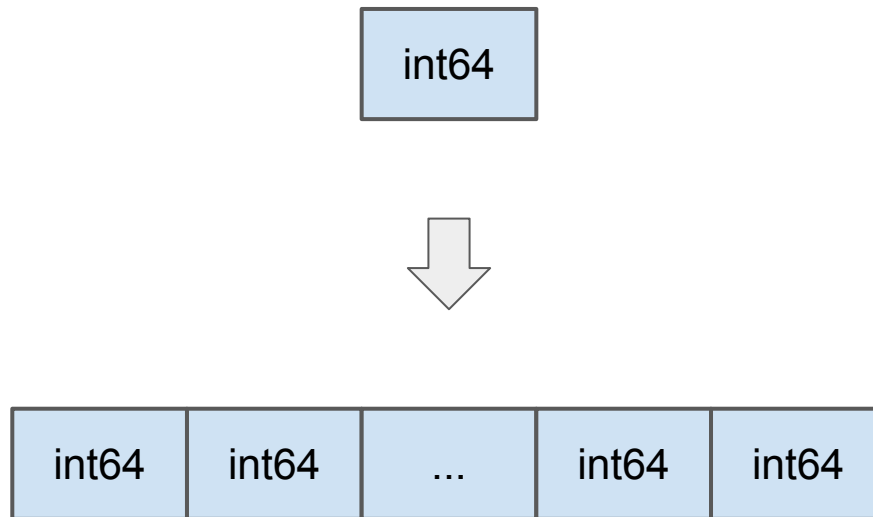
Атомарный счетчик

```
func main() {  
    cnt := int64(42)  
    go func() {  
        atomic.AddInt64(&cnt, 1)  
    }()  
    go func() {  
        atomic.AddInt64(&cnt, -1)  
    }()  
}
```

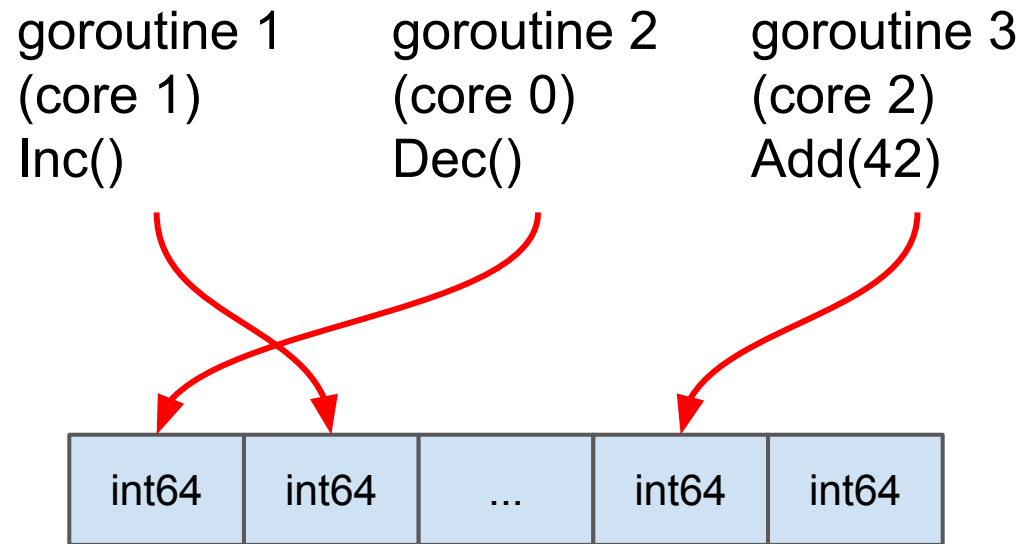
Атомарный счетчик

```
198      MOVQ      "".&cnt+40(SP), CX
199      ..... LOCK
200      ..... XADDQ  AX, (CX)|
201      MOVQ      "".&i+16(SP), AX
```

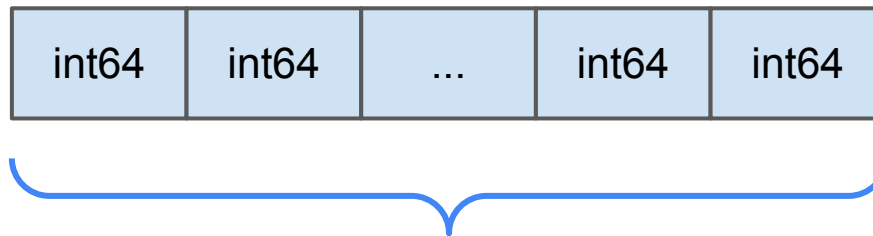
Атомарный счетчик с шардированием



Запись



Чтения



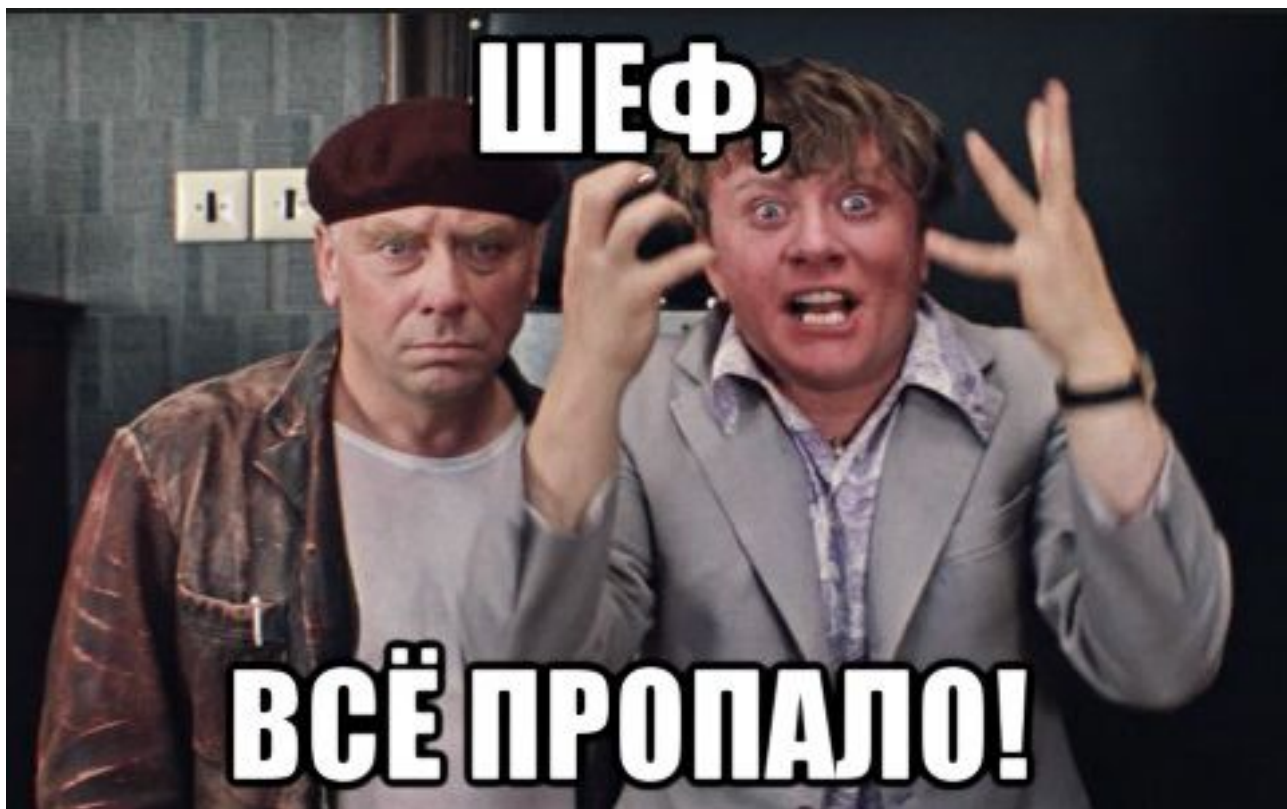
goroutine N
Value()

Как реализовать?

https://golang.org/doc/faq#no_goroutine_id

"Goroutines do not have names; they are just anonymous workers. They expose no unique identifier, name, or data structure to the programmer. Some people are surprised by this, expecting the go statement to return some item that can be used to access and control the goroutine later.

The fundamental reason goroutines are anonymous is so that the full Go language is available when programming concurrent code. By contrast, the usage patterns that develop when threads and goroutines are named can restrict what a library using them can do."



Вариант 1

- CPUID, x86
- getpid(2)

Вариант 2

Use the ~~for~~ `sync.Pool` Luke

P.S.

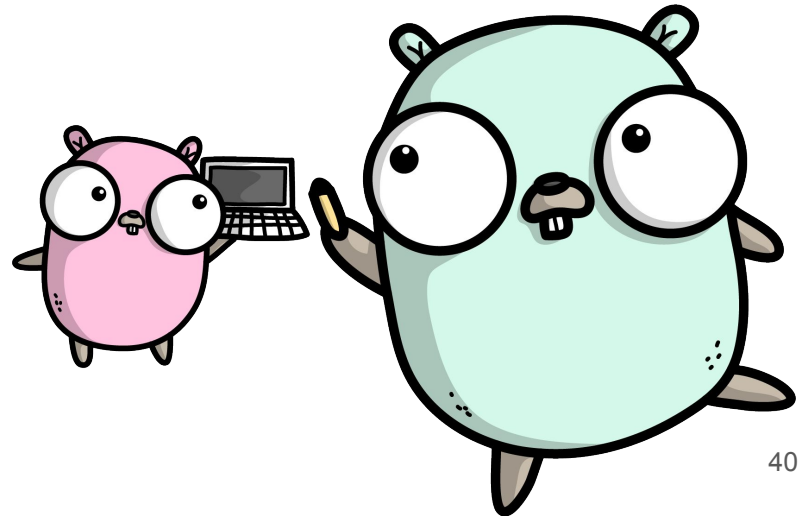
1. `sync.Pool` использует thread-local пулы
2. GC в Go не перемещает объекты

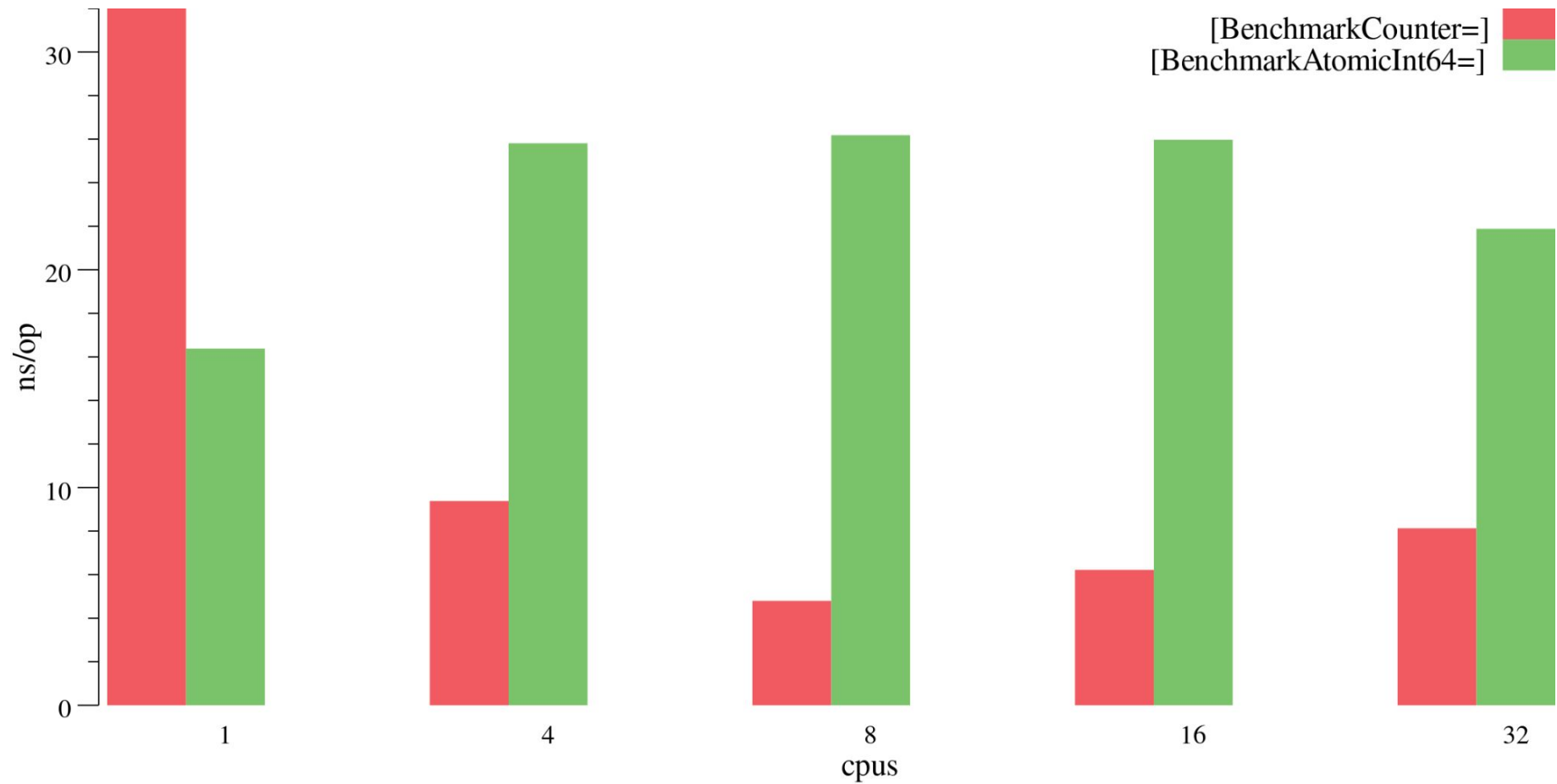
Вариант 2

```
type ptoken struct {
    idx uint32
}

func (c *Counter) Add(delta int64) {
    t, ok := pool.Get().(*ptoken)
    if !ok {
        t = new(ptoken)
        t.idx = uint32(hash64(uintptr(unsafe.Pointer(t))) & (cshards - 1))
    }
    // ...
    pool.Put(t)
}
```

Делаем замер?





Concurrent Inc/Dec, a value read on each 10,000 modification

xsync.Counter vs. atomic counter

Плюсы:

- Масштабируемость Inc/Dec

Минусы:

- Объем занимаемой памяти
- Чтение обходится "дорого"

sync.RWMutex

```
func main() {  
    var mu sync.RWMutex  
    for i := 0; i < 10; i++ {  
        go func() {  
            mu.RLock()  
            // ...reader critical section  
            mu.RUnlock()  
        }()  
    }  
    go func() {  
        mu.Lock()  
        // ...writer critical section  
        mu.Unlock()  
    }()  
}
```

sync.RWMutex изнутри

```
type RWMutex struct {  
    w          Mutex  
    readerCount int32  
    // ...  
}  
  
func (rw *RWMutex) RLock() {  
    // ...  
    // Fast path for readers:  
    if atomic.AddInt32(&rw.readerCount, 1) < 0 {  
        runtime_SemacquireMutex(&rw.readerSem, false, 0)  
    }  
    // ...  
}
```

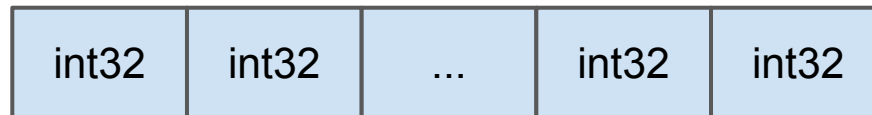
Можно ли быстрее, чем `sync.RWMutex`?

BRAVO (Biased Locking for Reader-Writer Locks),
2019, D.Dice, A.Kogan, Oracle Labs

Обертка поверх Reader-Writer Lock, в частности
`sync.RWMutex`.

BRAVO

readers



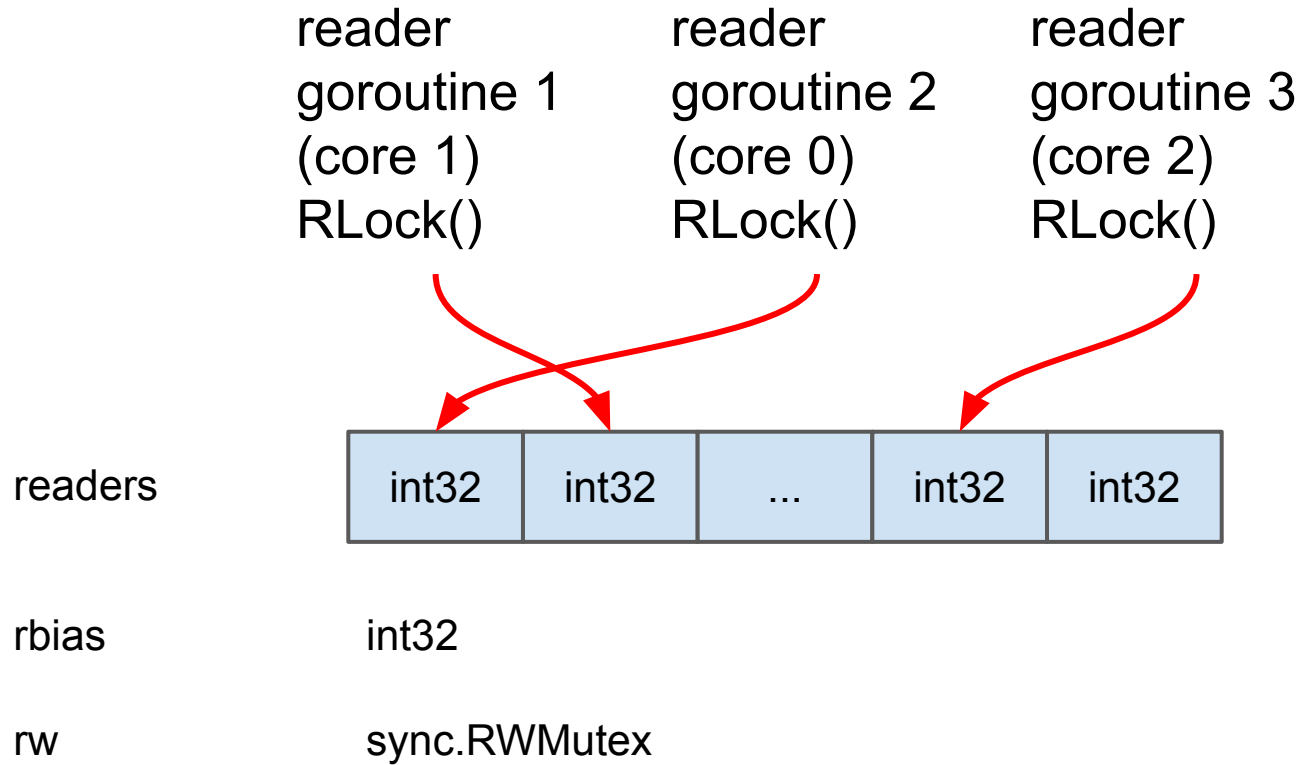
rbias

int32

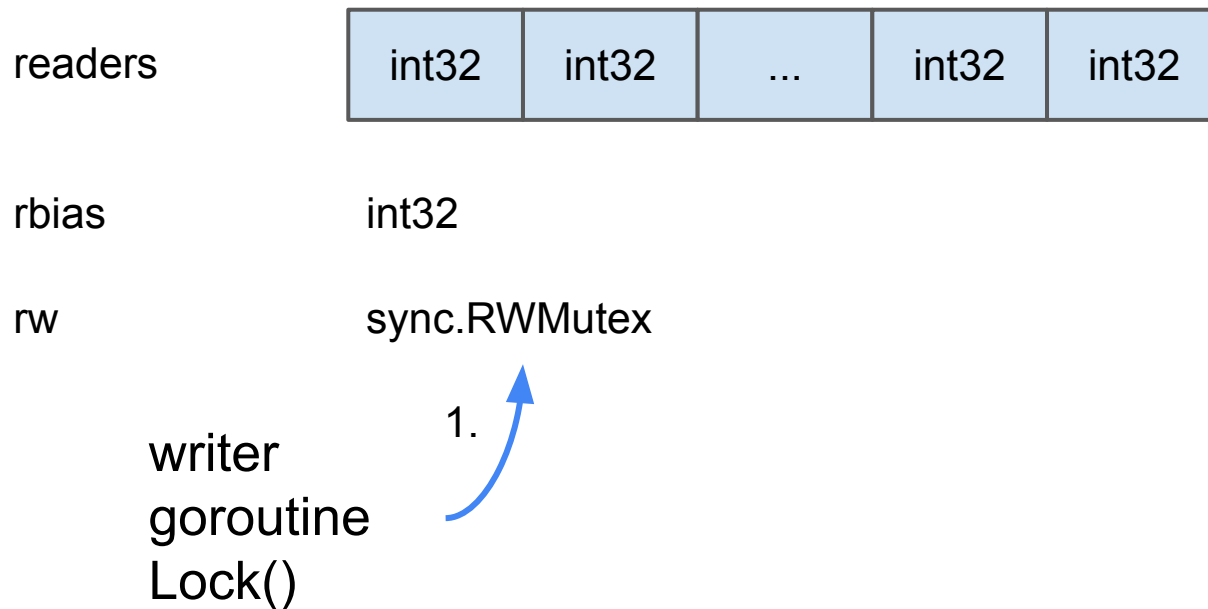
rw

sync.RWMutex

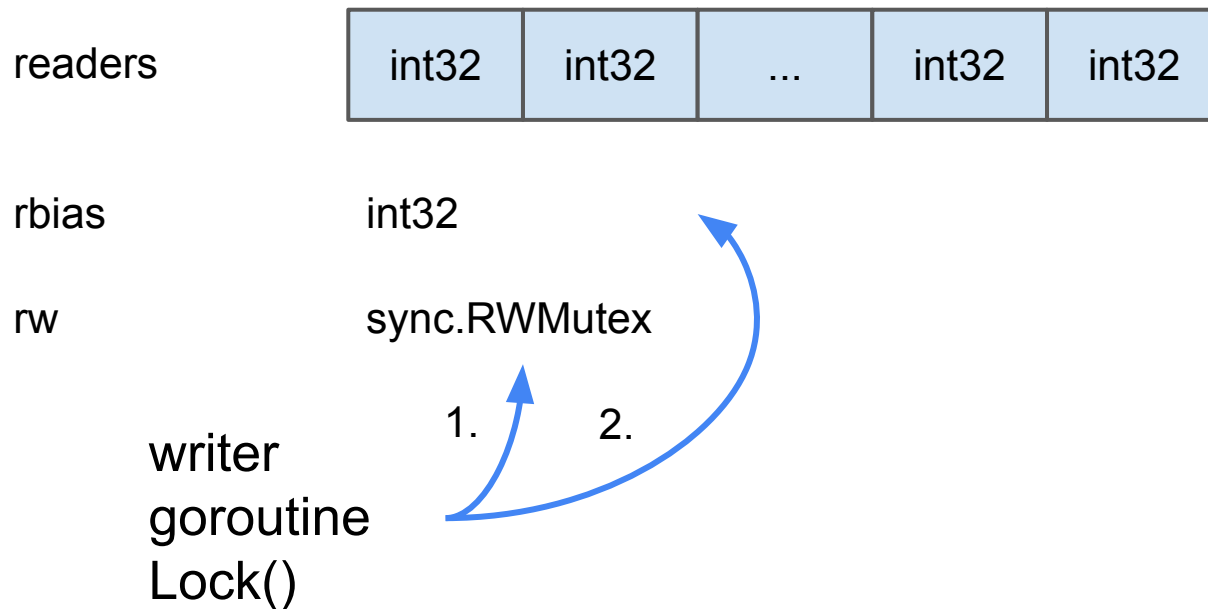
BRAVO



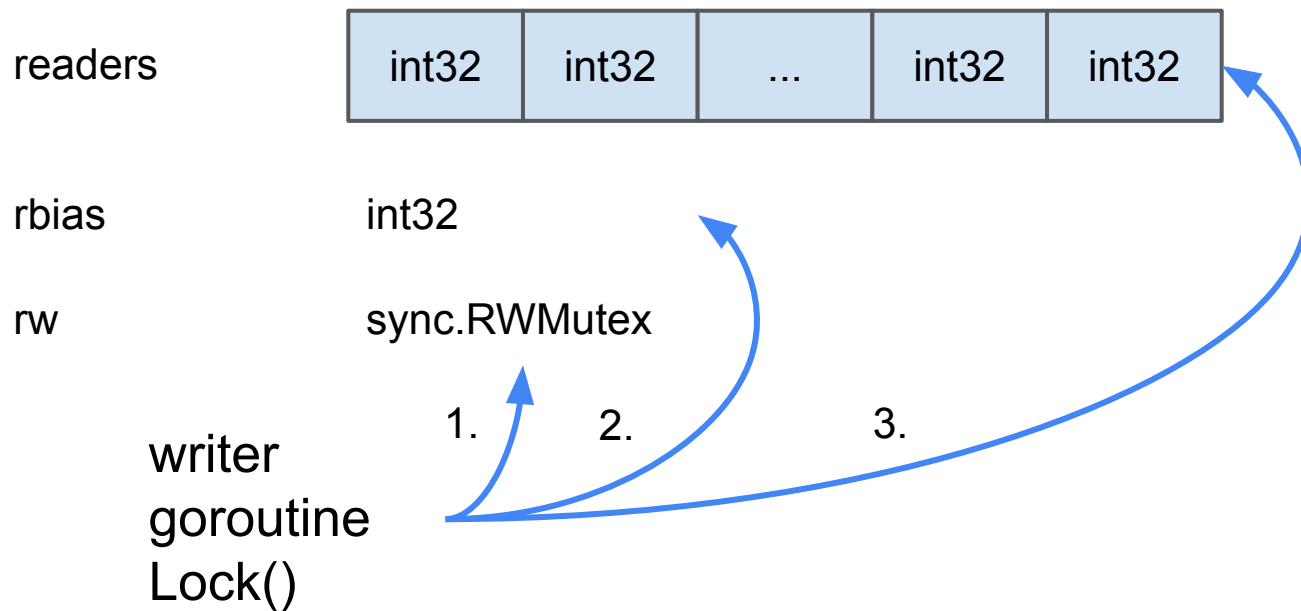
BRAVO



BRAVO



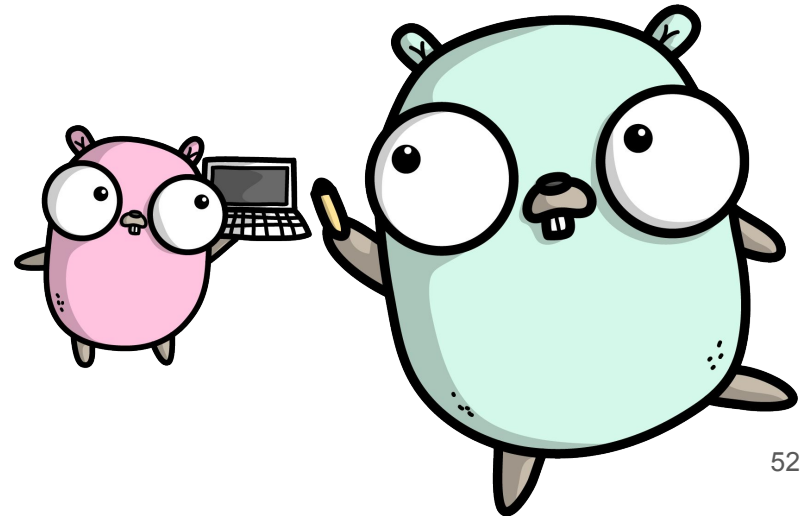
BRAVO

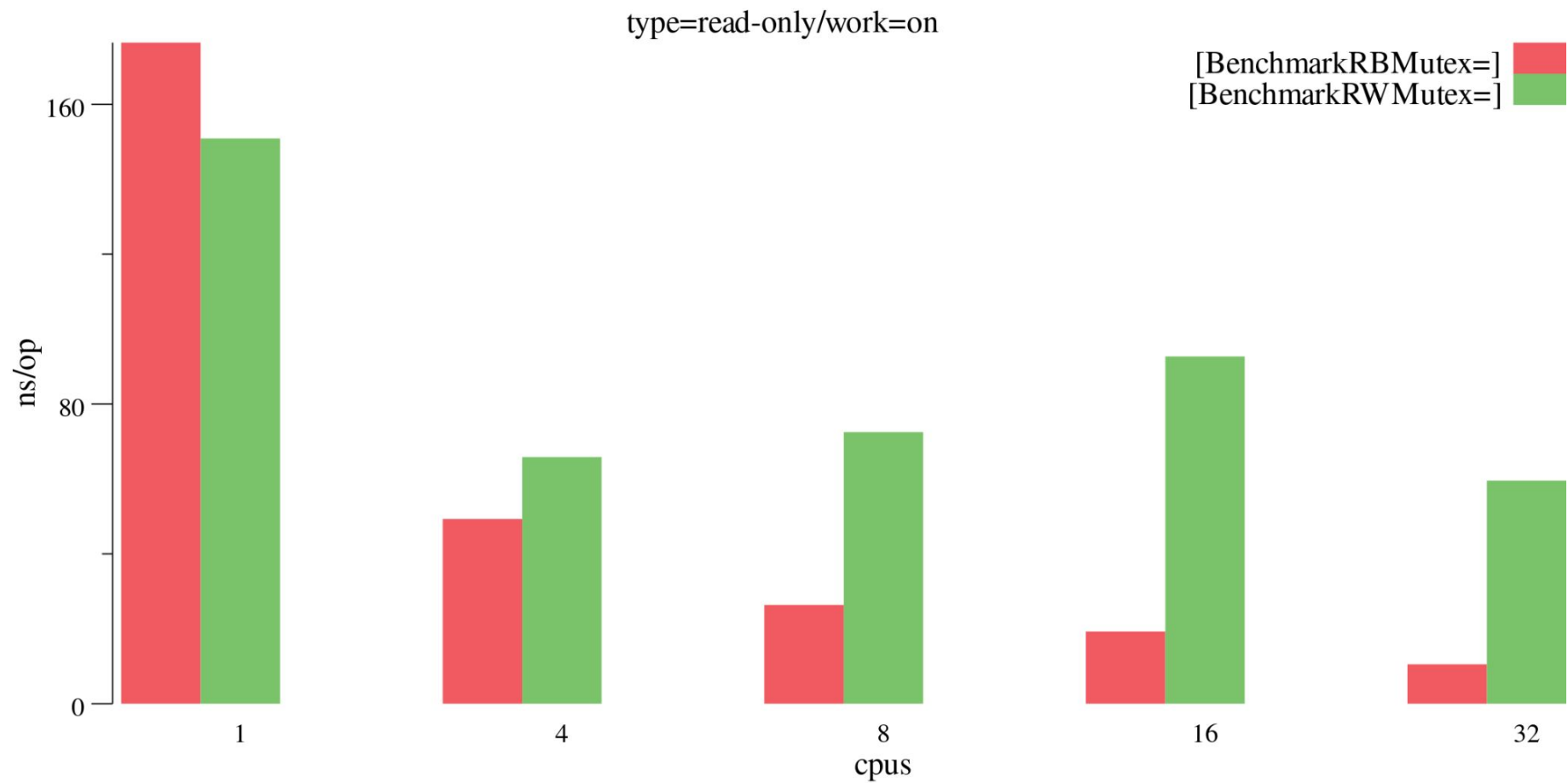


xsync.RBMutex

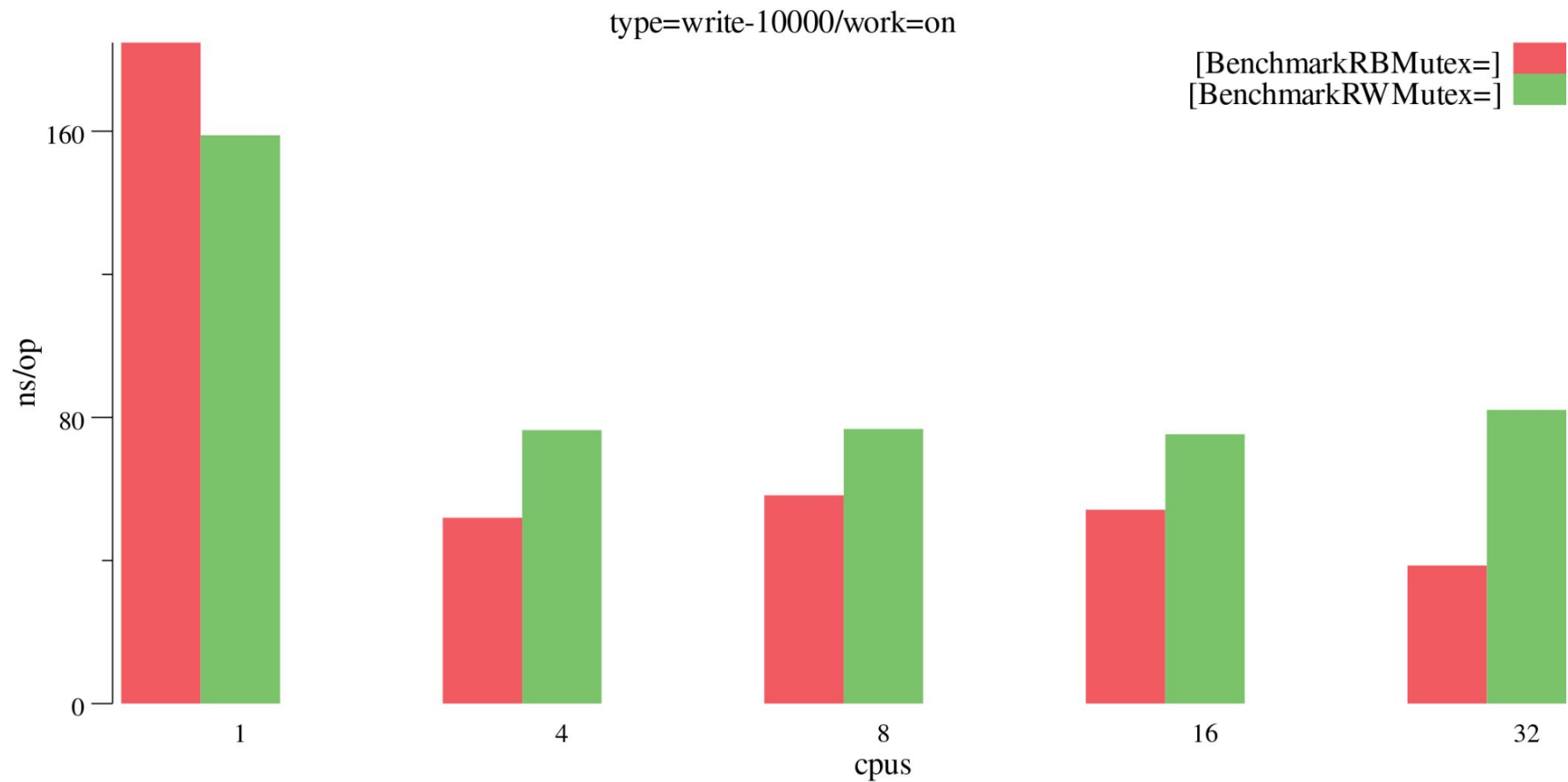
```
func main() {  
    var mu xsync.RBMutex  
    for i := 0; i < 10; i++ {  
        go func() {  
            t := mu.RLock() // <--- reader token  
            // ...  
            mu.RUnlock(t)  
        }()  
    }  
    go func() {  
        mu.Lock()  
        // ...  
        mu.Unlock()  
    }()  
}
```

Делаем замер?





Reader locks only, some work in the critical section



Writer locks on each 10,000 iteration, some work in the critical section

xsync.RBMutex vs. sync.RWMutex

Плюсы:

- Масштабируемость RLock/RUnlock

Минусы:

- Объем занимаемой памяти
- Lock/Unlock обходятся "дорого"

Каналы

```
func main() {  
    ch := make(chan int, 42)  
    go func() {  
        for i := 0; i < 42; i++ {  
            ch <- i  
        }  
    }()  
    go func() {  
        for i := range ch {  
            // ...  
        }  
    }()  
}
```


Каналы изнутри

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool
{
    // ...

    lock(&c.lock) // <--- Good old mutex.

    if c.closed != 0 {
        unlock(&c.lock)
        panic(plainError("send on closed channel"))
    }

    if sg := c.recvq.dequeue(); sg != nil {
        send(c, sg, ep, func() { unlock(&c.lock) }, 3)
        return true
    }

    // ...
}
```

Можно ли быстрее, чем каналы?

Каналы - multiple producer single consumer (MPMC) очередь.

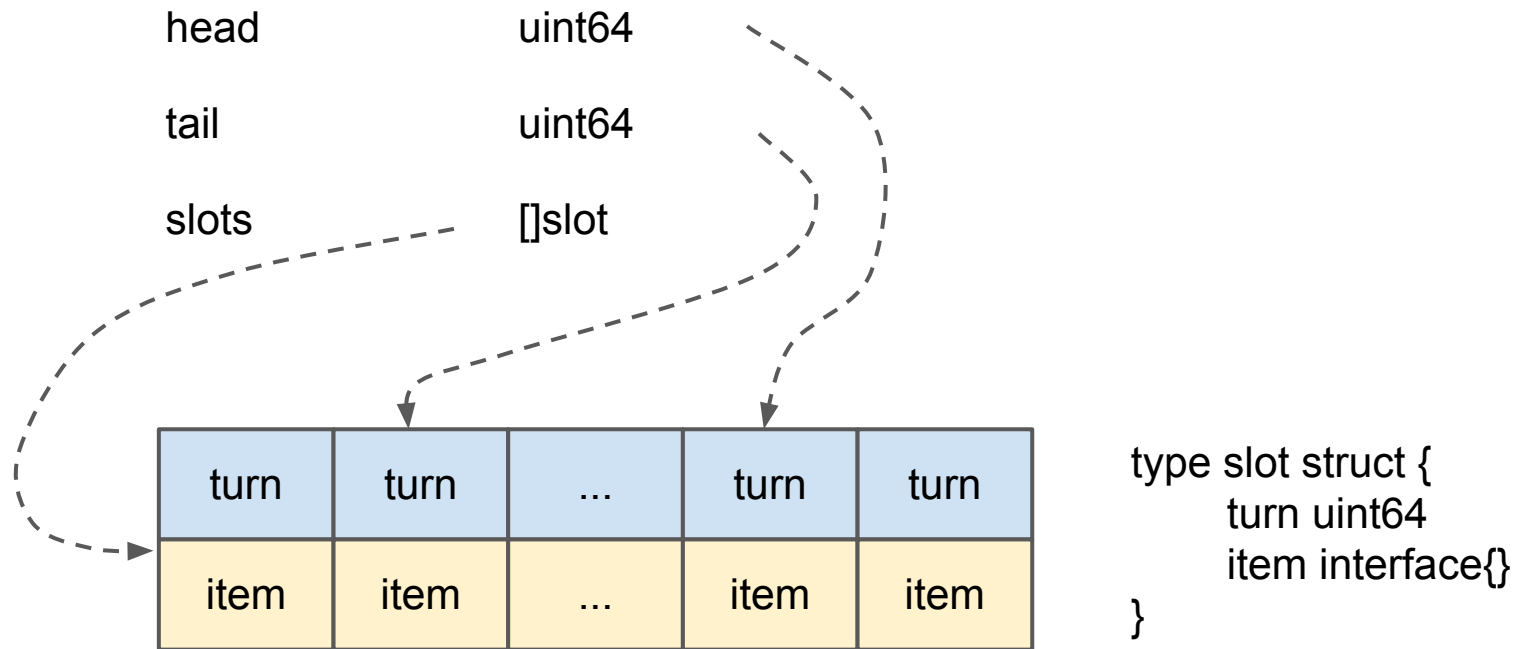
Можно ли быстрее, чем каналы?

[MPMCQueue](#), C++ библиотека, Frostbite game engine.

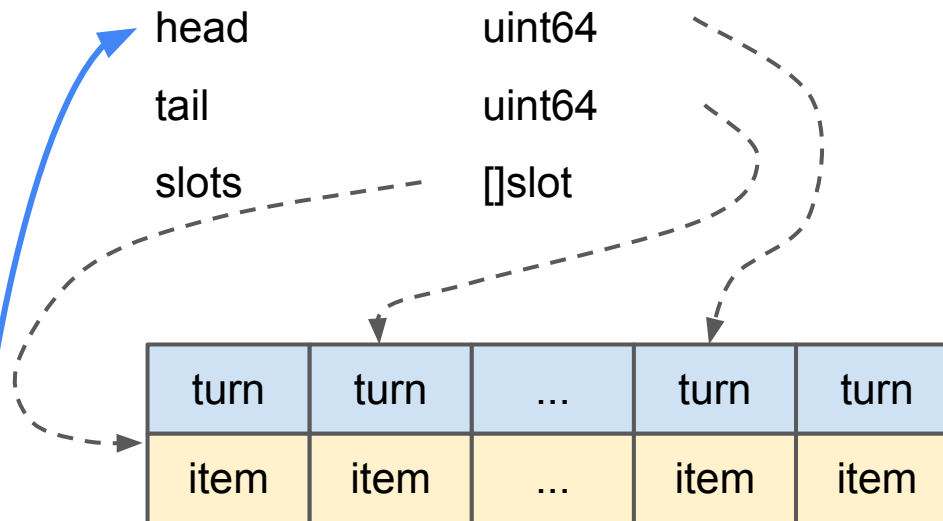
Основана на D.Vyukov's MPMC bounded queue:

- array-based
- causal FIFO
- w/o priorities
- fails on overflow
- does not require GC

MPMCQueue



MPMCQueue: поставщики

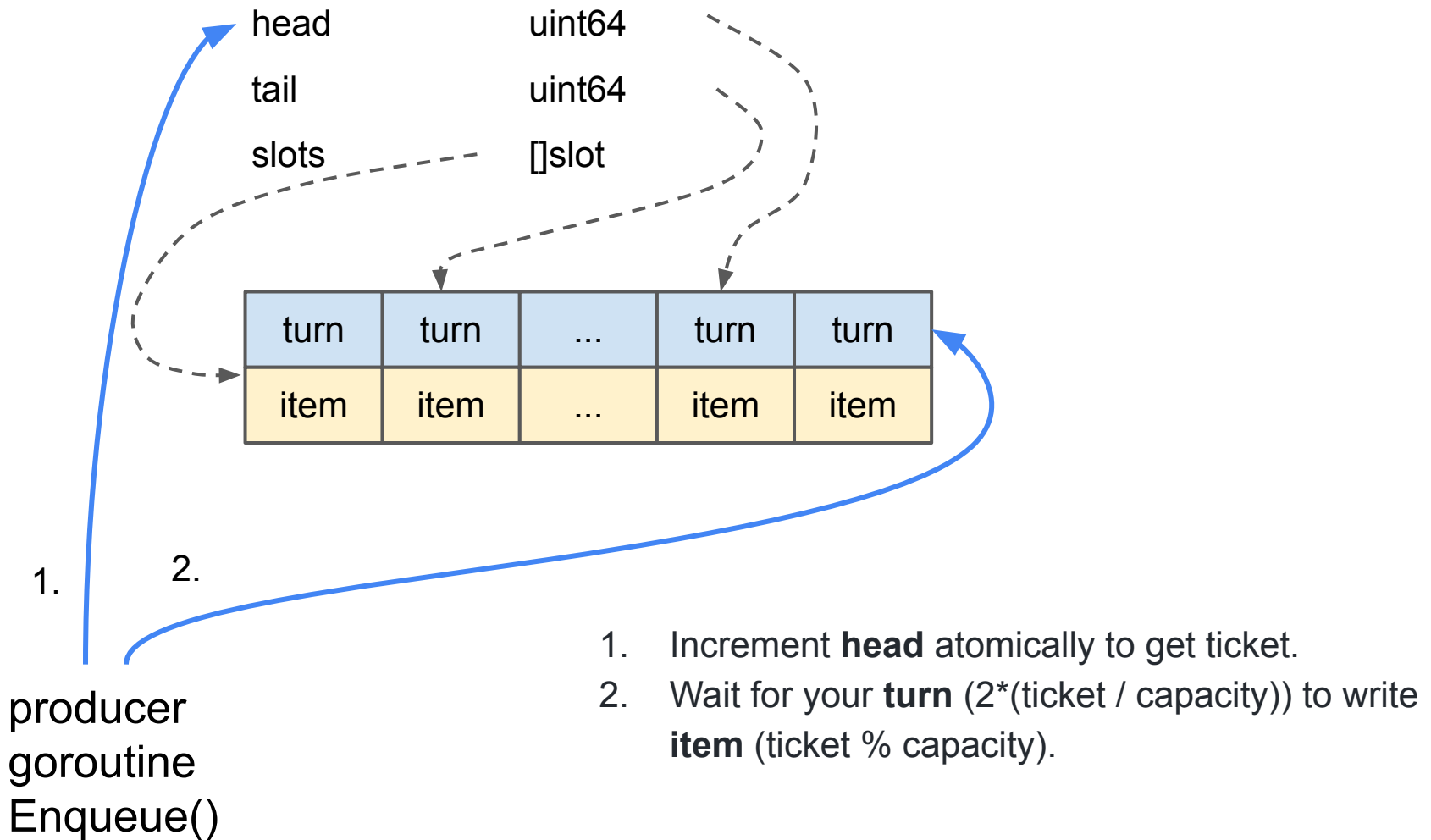


1.

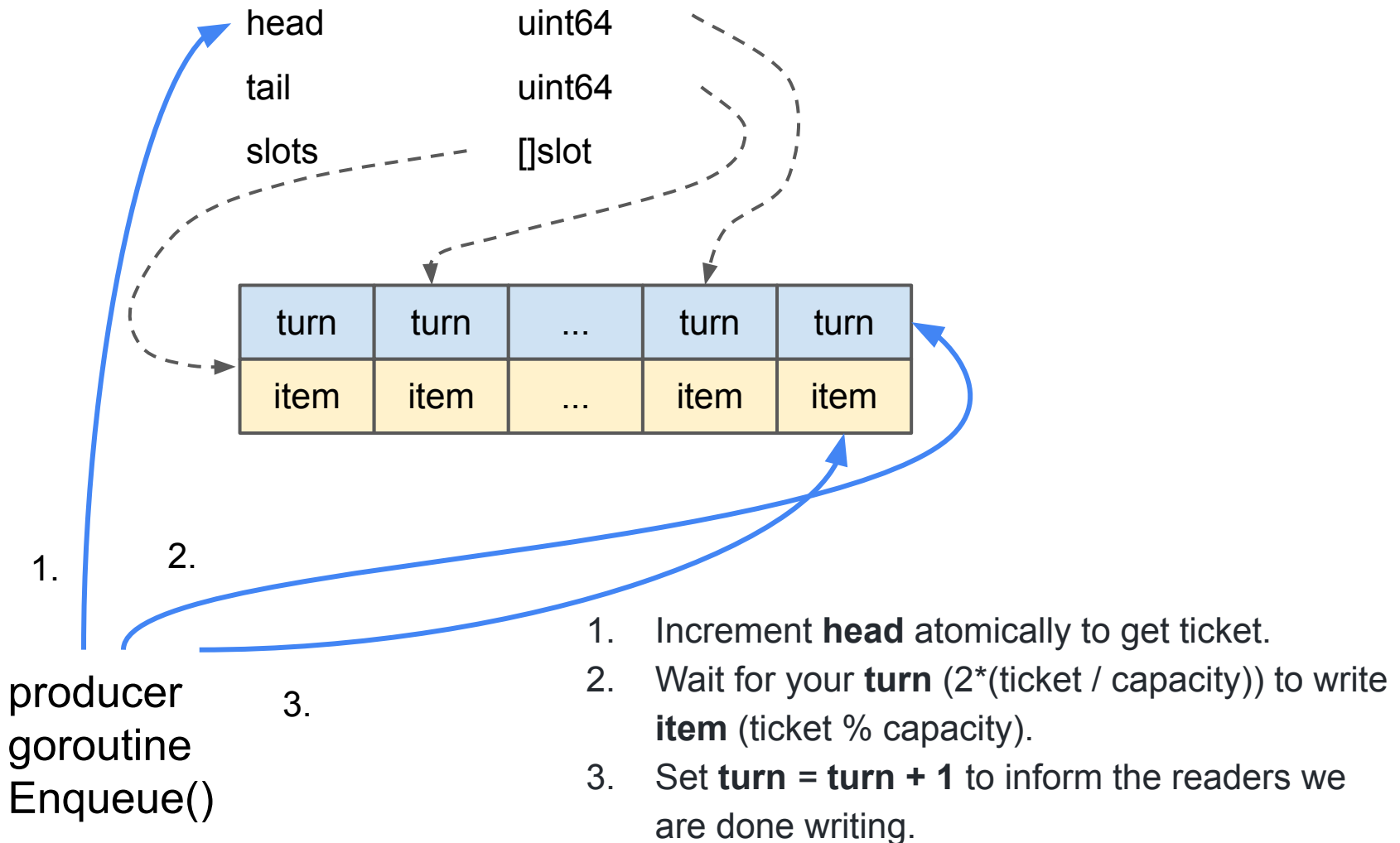
producer
goroutine
Enqueue()

1. Increment **head** atomically to get ticket.

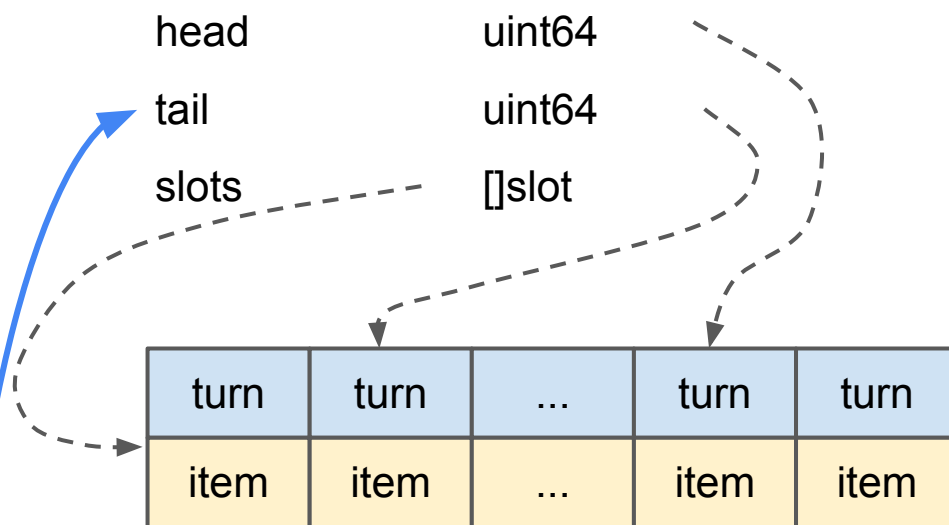
MRMCSQueue: поставщики



MPMCQueue: поставщики



MPMCQueue: потребители

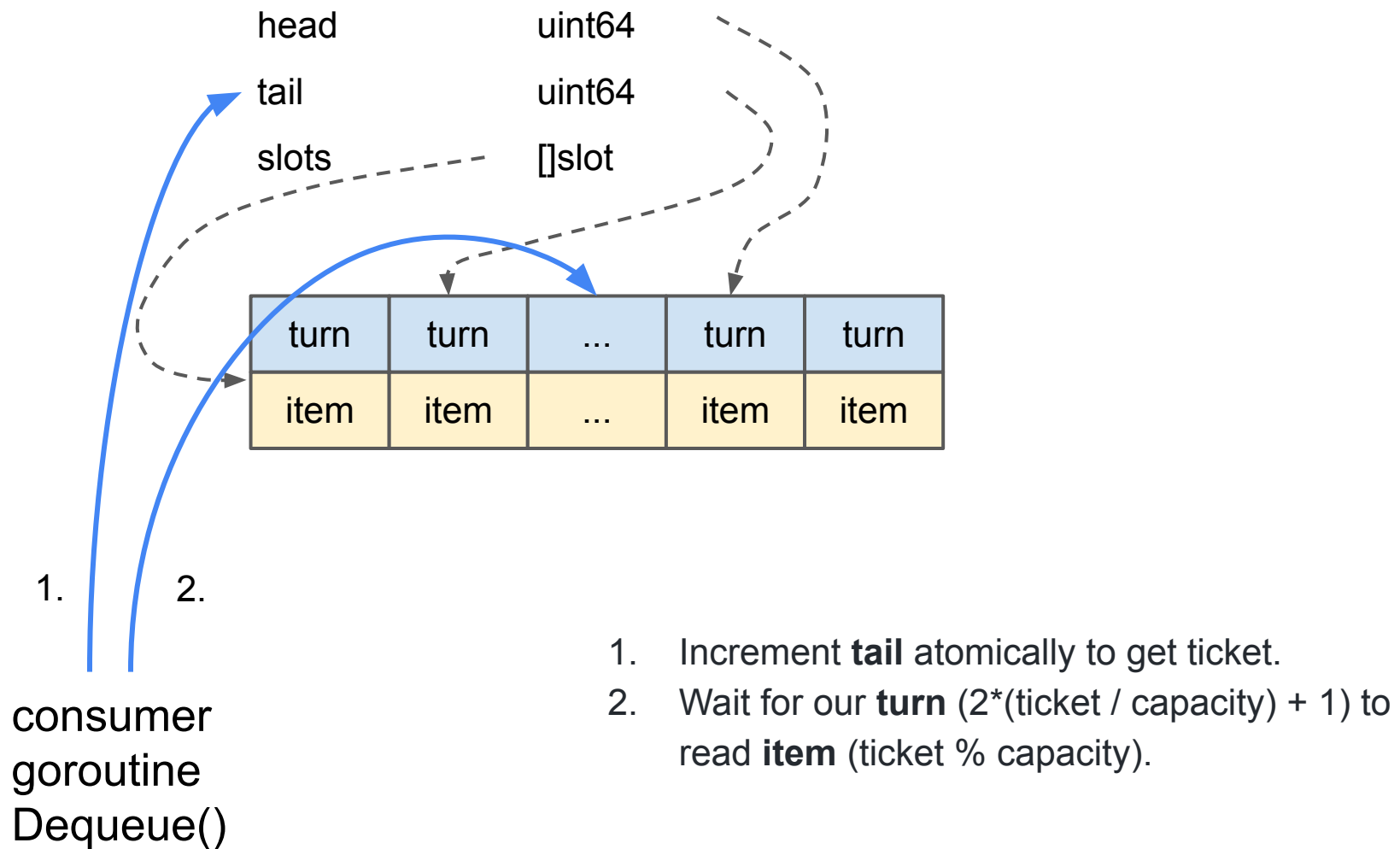


1.

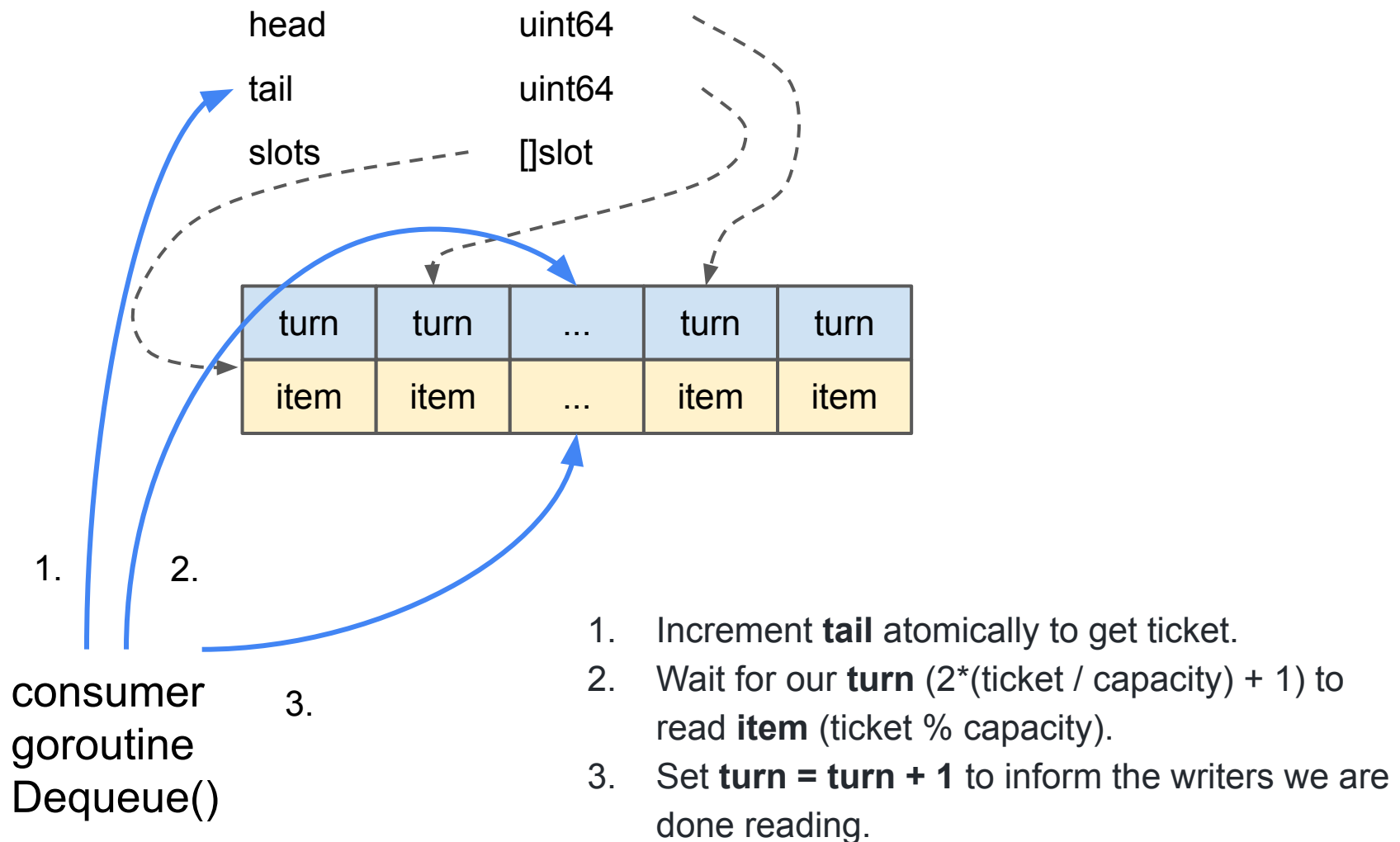
consumer
goroutine
Dequeue()

1. Increment **tail** atomically to get ticket.

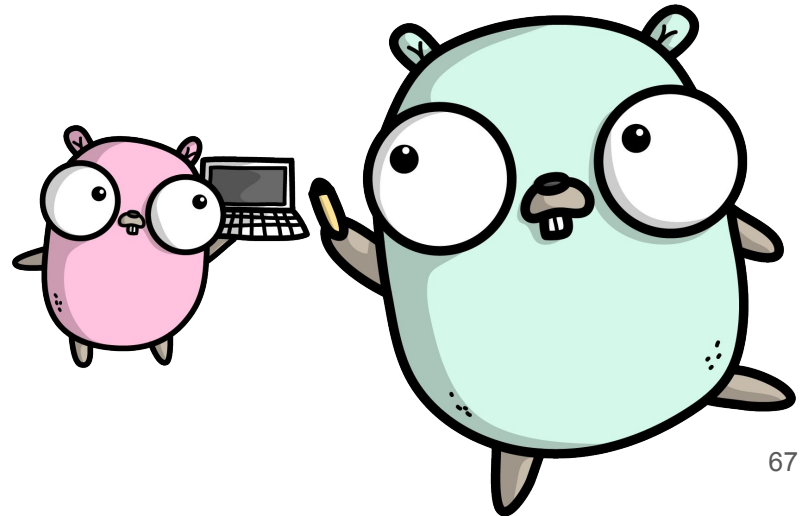
MPMCQueue: потребители

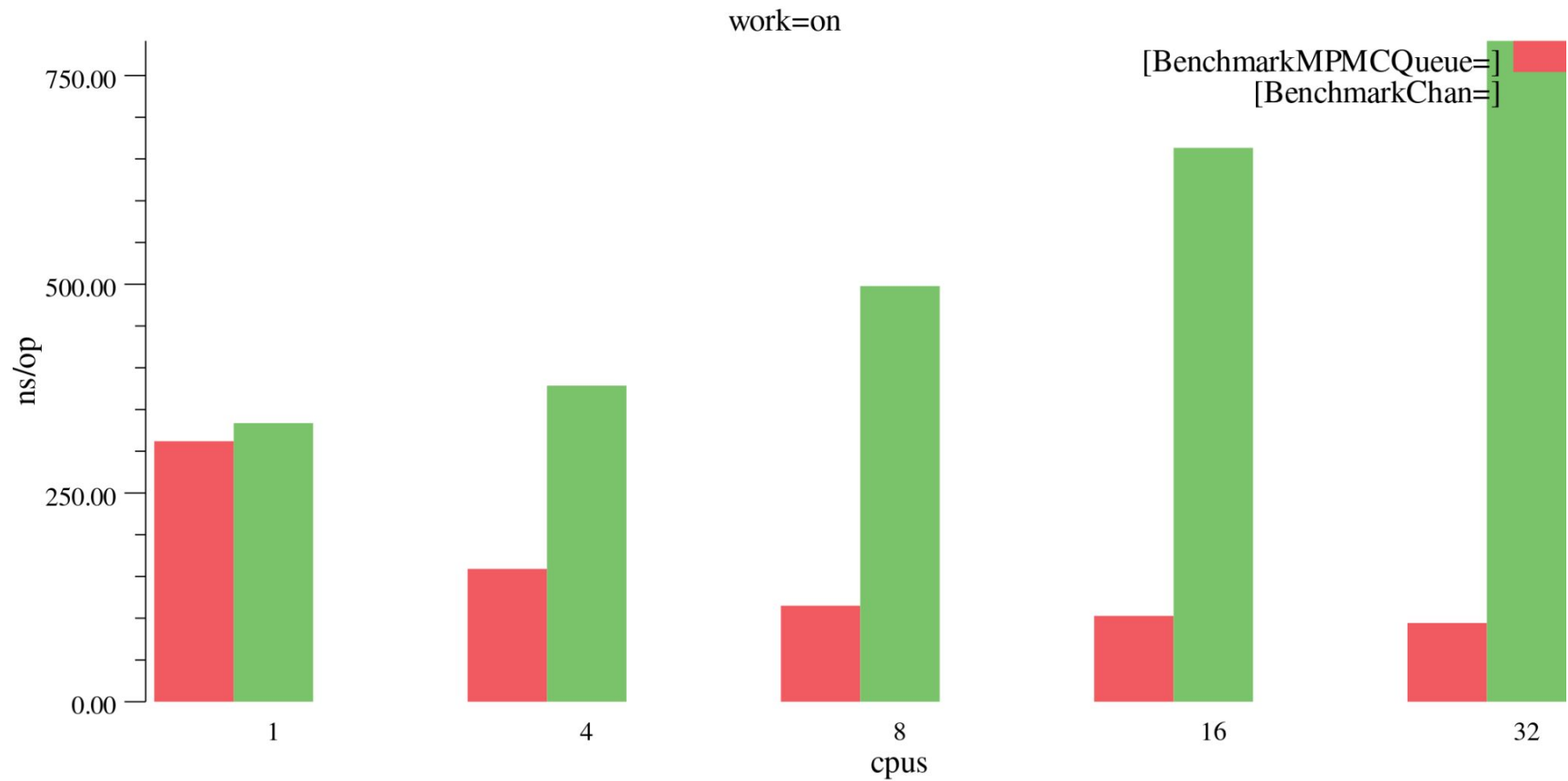


MPMCQueue: потребители



Делаем замер?





Concurrent producers and consumers (1:1), queue/channel size 1,000, some work

xsync.MPMCSQueue vs. chan

Плюсы:

- Масштабируемость

Минусы:

- Объем занимаемой памяти

map + Mutex/RWMutex

```
func main() {  
    type rormap struct {  
        data map[int]int  
        mu    sync.RWMutex  
    }  
  
    m := rormap{data: make(map[int]int)}  
    for i := 0; i < 10; i++ {  
        go func(nreader int) {  
            m.mu.RLock()  
            j, ok := m.data[nreader]  
            // ...  
            m.mu.RUnlock()  
        }(i)  
    }  
    // ...  
}
```

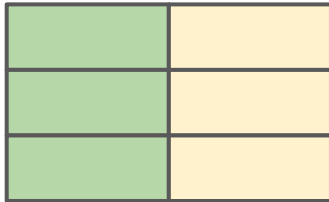
map + Mutex/RWMutex

- map + Mutex - не масштабируется
- map + RWMutex - запись не масштабируется, масштабирование чтений упирается в RWMutex

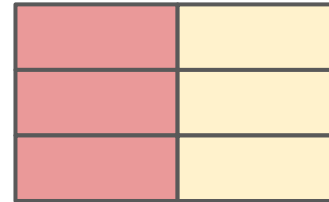
sync.Map

```
func main() {  
    var m sync.Map  
    for i := 0; i < 10; i++ {  
        go func(nreader int) {  
            j, ok := m.Load(nreader)  
            // ...  
        }(i)  
    }  
    go func() {  
        m.Store(0, 42)  
    }()  
}
```


sync.Map изнутри

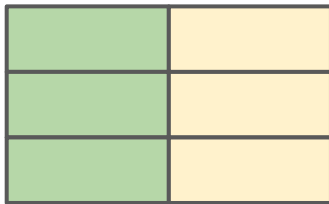


read map[interface{}]*entry
amended bool

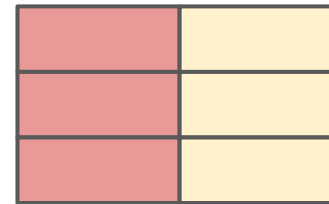


dirty map[interface{}]*entry
mu sync.Mutex

sync.Map: чтение



read map[interface{}]*entry
amended bool



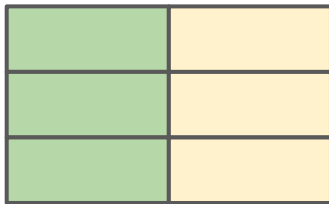
dirty map[interface{}]*entry
mu sync.Mutex

1.

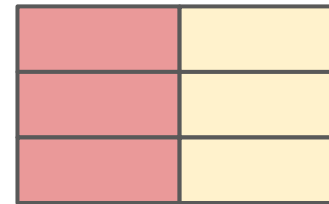


reader
goroutine
Load(42)

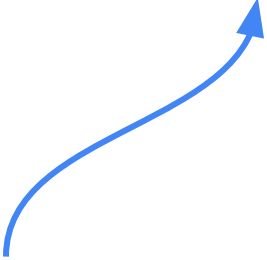
sync.Map: чтение

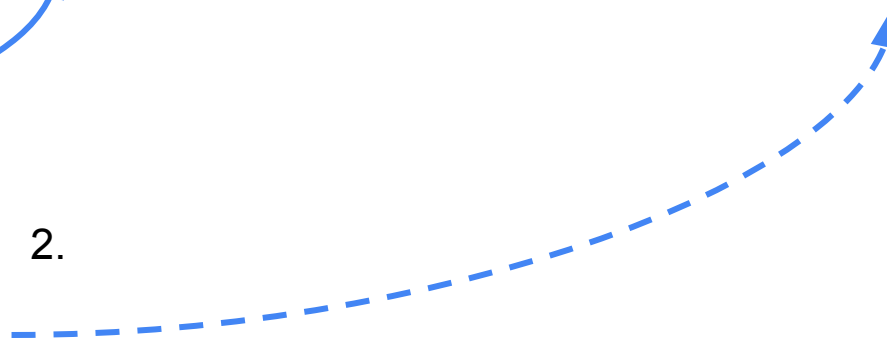


read map[interface{}]*entry
amended bool

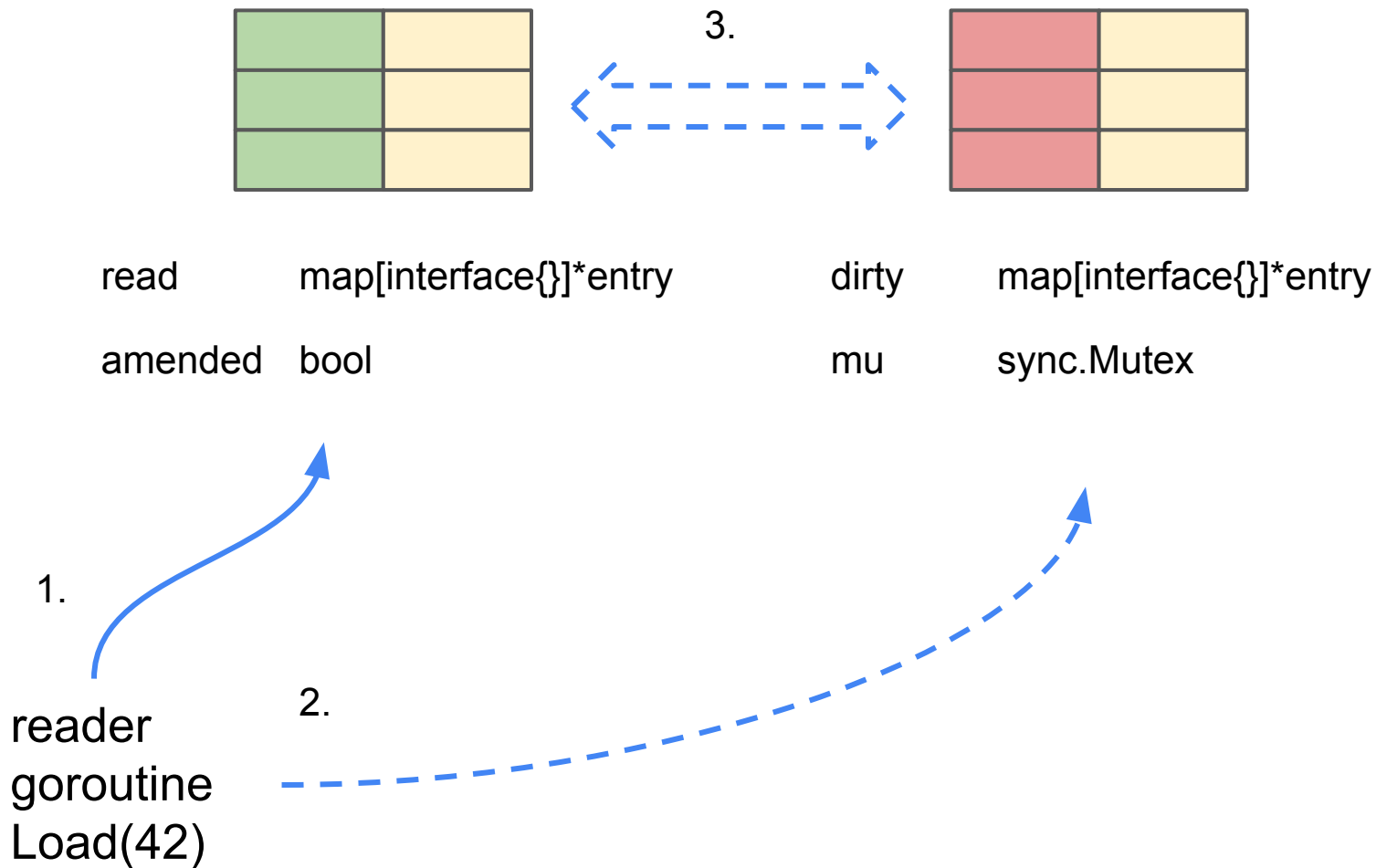


dirty map[interface{}]*entry
mu sync.Mutex

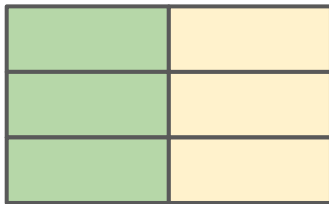
1. 
reader
goroutine
Load(42)

2. 

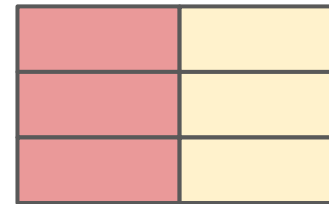
sync.Map: чтение



sync.Map: запись

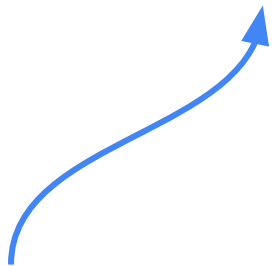


read map[interface{}]*entry
amended bool



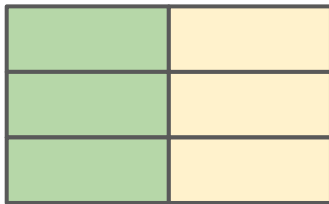
dirty map[interface{}]*entry
mu sync.Mutex

1.

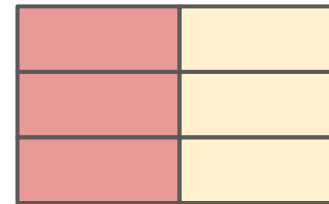


writer
goroutine
Store(42, 0)

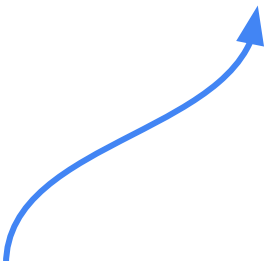
sync.Map: запись

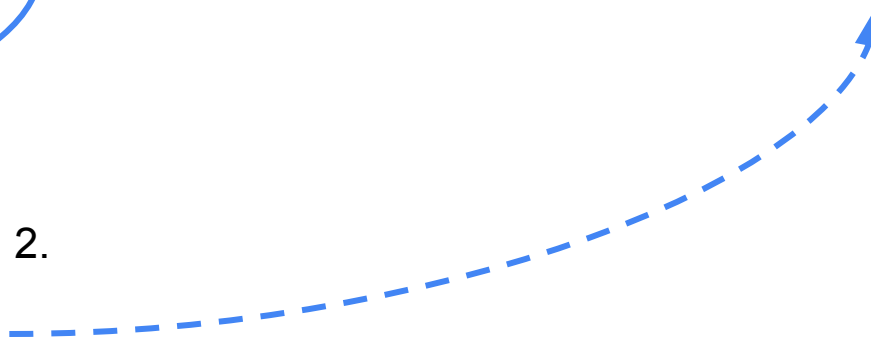


read map[interface{}]*entry
amended bool



dirty map[interface{}]*entry
mu sync.Mutex

1. 
writer
goroutine
Store(42, 0)

2. 

Можно ли быстрее?

Один из вариантов - sharded map + RWMutex.

Например:

<https://github.com/orcaman/concurrent-map>

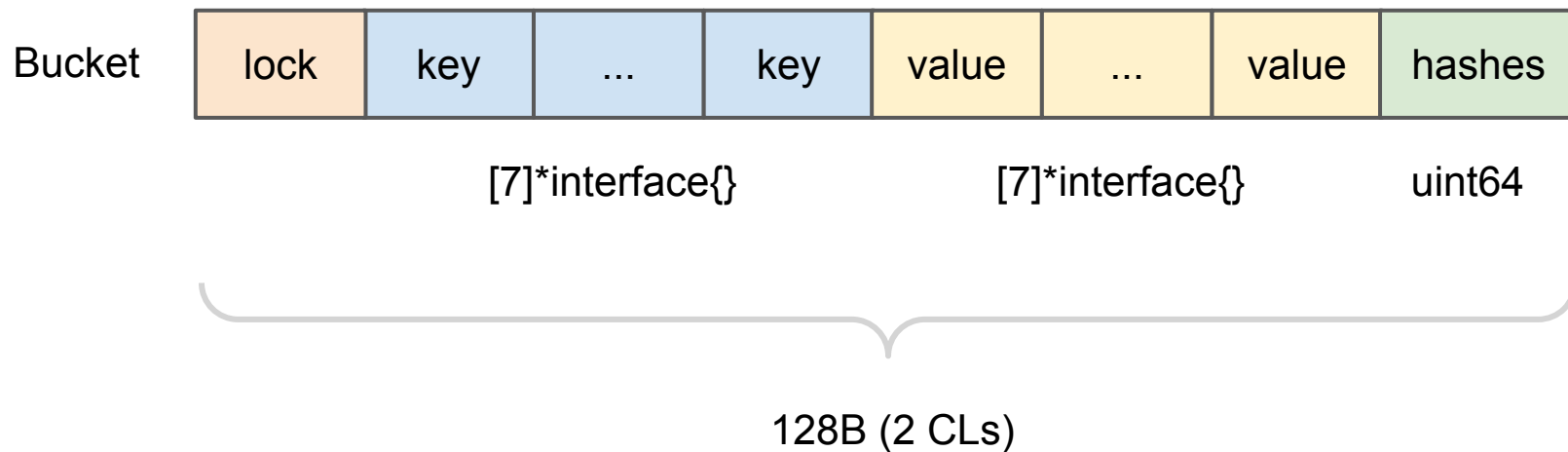
Основной минус - чтения не масштабируются линейно.

Можно ли еще быстрее?

Cache-Line Hash Table (CLHT)

- Чтения масштабируются линейно (atomic snapshot)
- Ячейки размером в кэш-линию
- Lock-based и lock-free вариации
- С цепочками и без

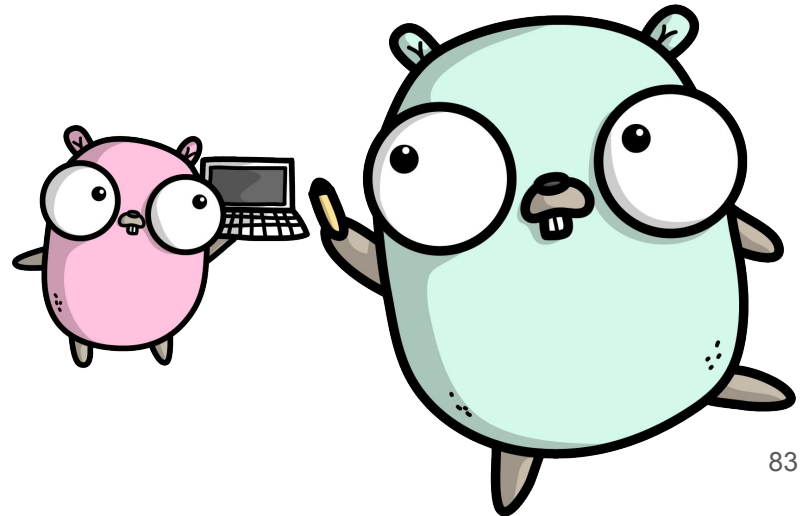
CLHT: ячейки

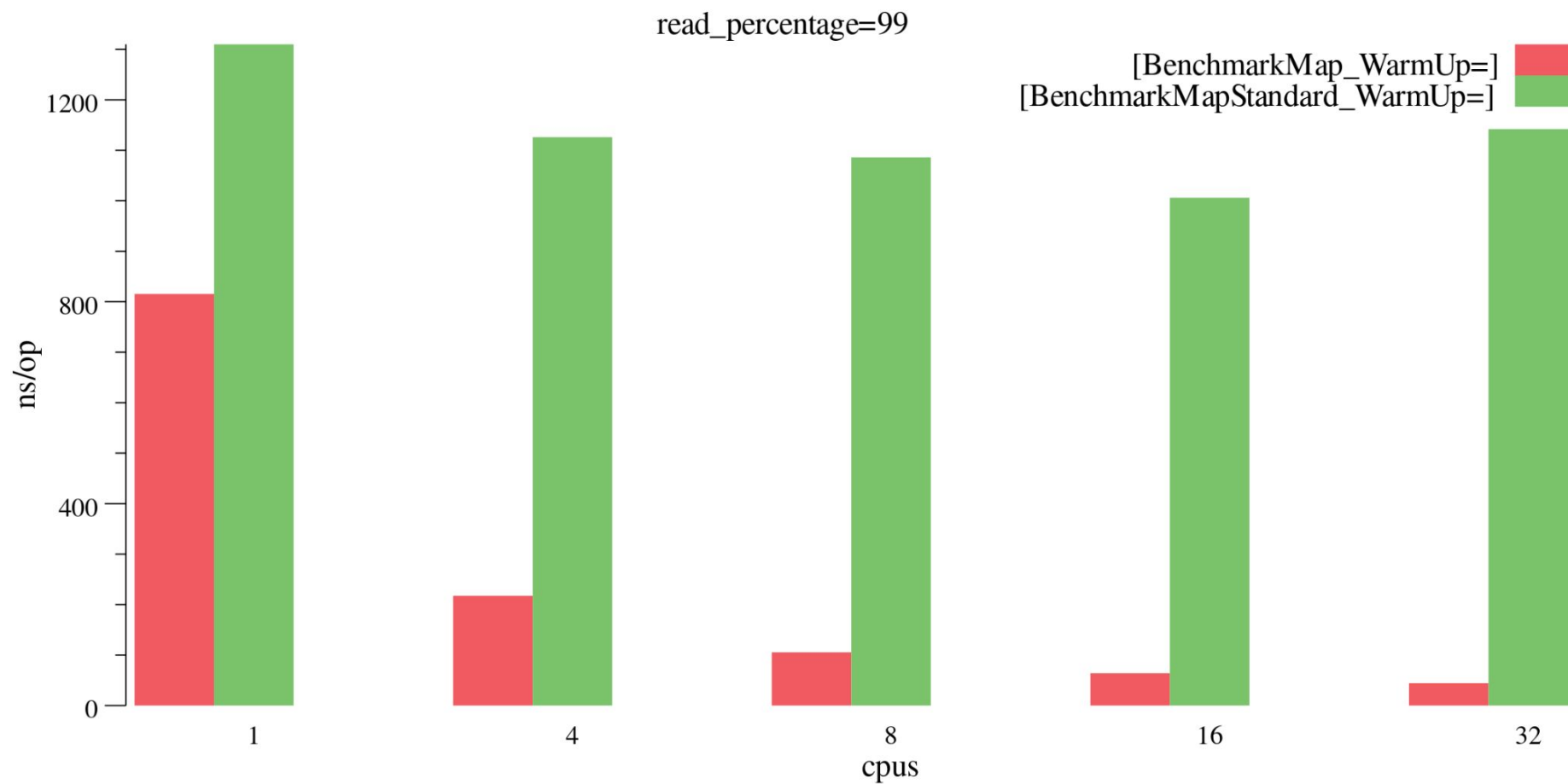


CLHT: atomic snapshot

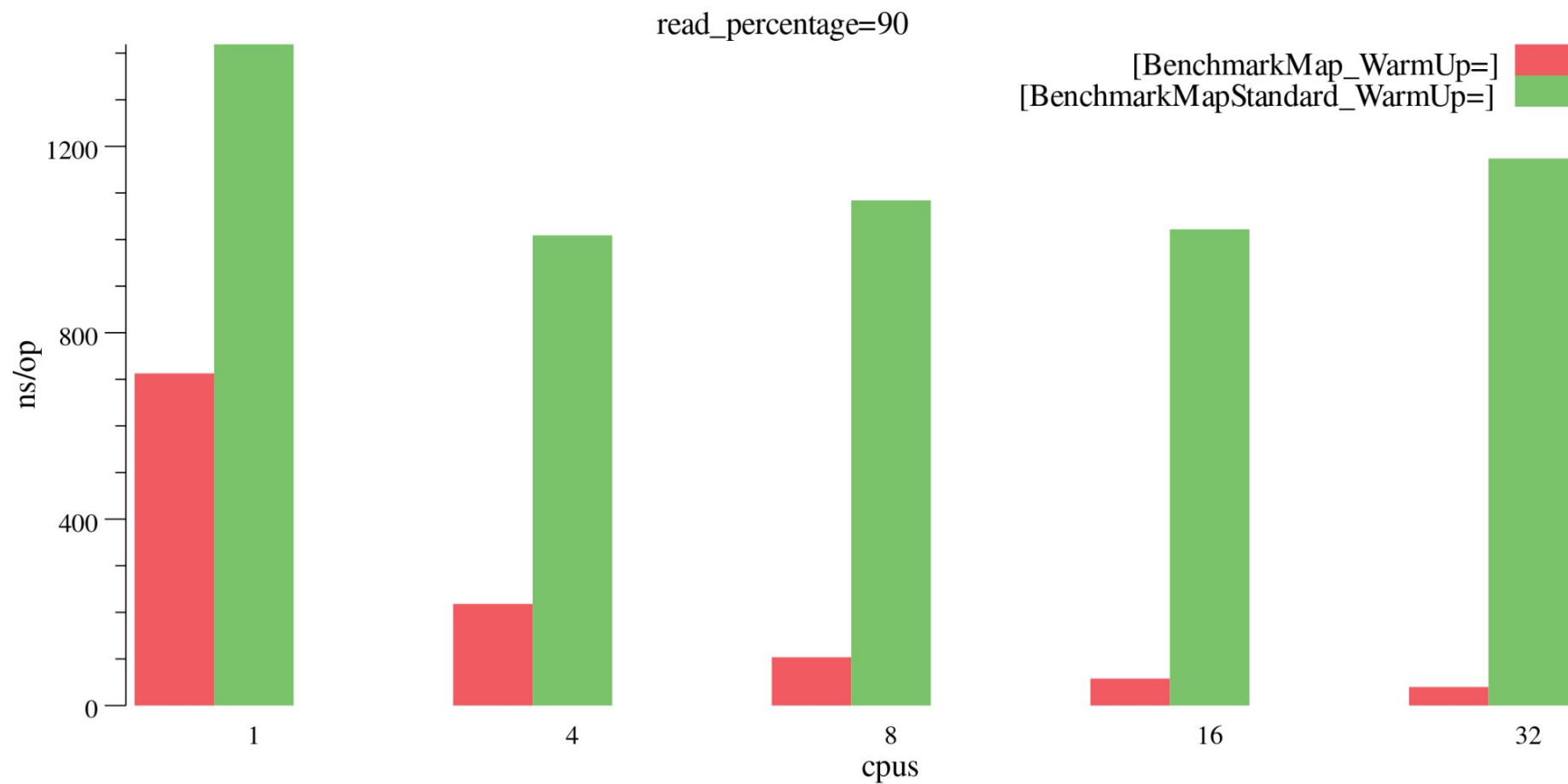
```
func (m *Map) Load(key string) (value interface{}, ok bool) {  
    // ...  
    for i := 0; i < entriesPerMapBucket; i++ {  
        atomic_snapshot:  
        vp := atomic.LoadPointer(&b.values[i])  
        kp := atomic.LoadPointer(&b.keys[i])  
        if kp != nil && vp != nil {  
            if key == derefKey(kp) {  
                if uintptr(vp) == uintptr(atomic.LoadPointer(&b.values[i])) {  
                    return derefValue(vp), true // Atomic snapshot succeeded.  
                }  
                goto atomic_snapshot // Concurrent update/remove.  
            }  
        }  
    }  
    return nil, false  
}
```

Делаем замер?

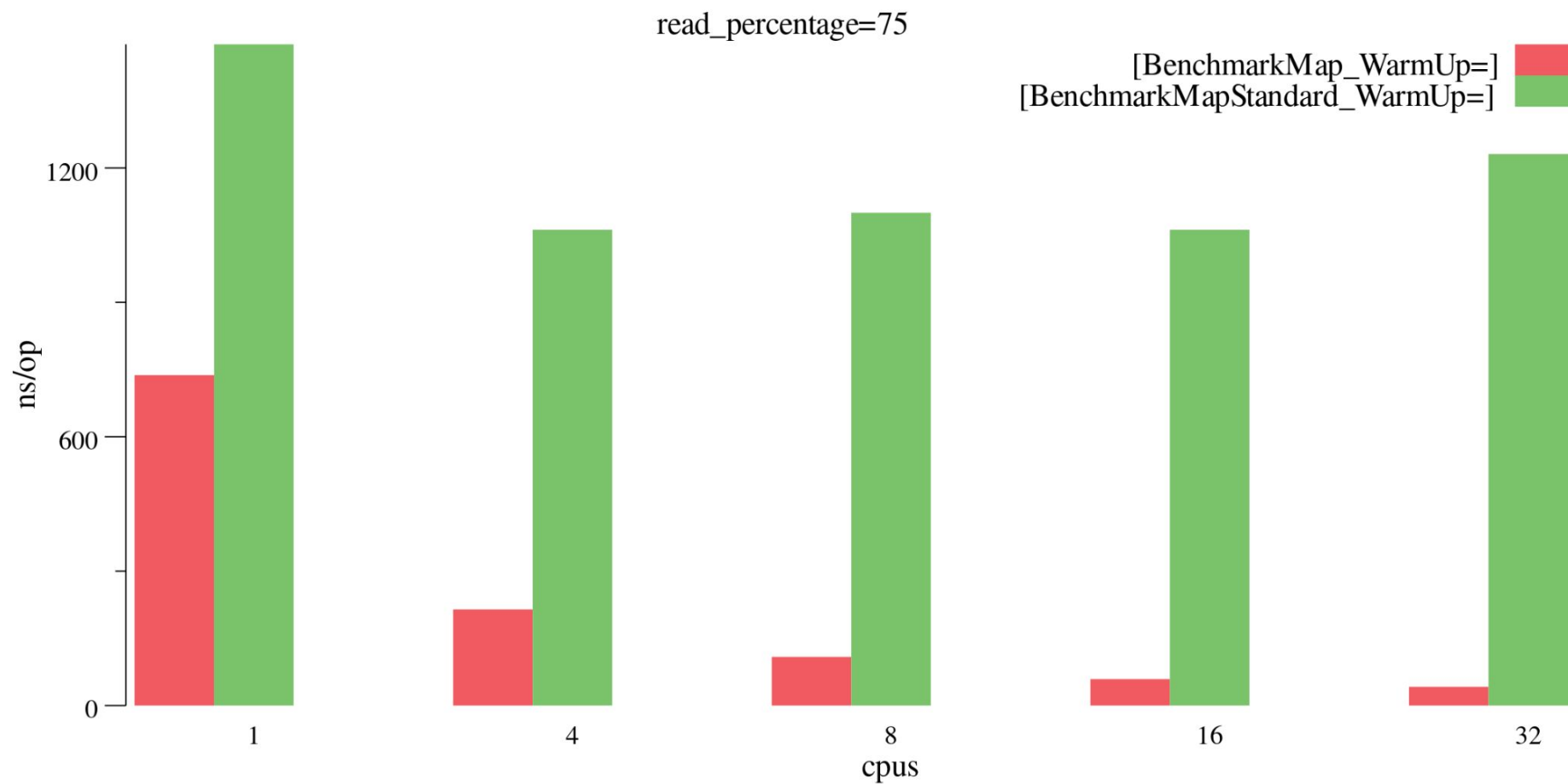




1M entries with warm-up, 99% Gets, 0.5% Stores, 0.5% Deletes



1M entries with warm-up, 90% Gets, 5% Stores, 5% Deletes



1M entries with warm-up, 75% Gets, 12.5% Stores, 12.5% Deletes

xsync.Map vs. sync.Map

Плюсы:

- Масштабируемость

Минусы:

- Поддержка только string ключей

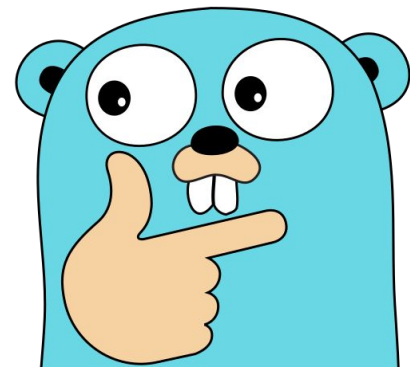
Core proposal - new version of sync.Map:

<https://github.com/golang/go/issues/47643>

Недостатки xsync.Map

- Заполненность до перестроения хеш-таблицы может быть не очень высокой
- Промежуточные `interface{}` структуры ведут к лишней нагрузке на GC
- Всегда можно выше, быстрее, сильнее

Выводы



Где найти код?

<https://github.com/puzpuzpuz/xsync>



Стоит брать к себе в кодовую базу?

Краткий ответ - нет.

P.S. Использование библиотеки, benchmark'и, issue и PR'ы всячески приветствуются.

Выводы

- Стандартные подходы не всегда оптимальны
- Писать быстрый многопоточный код - сложно
- Если все же пришлось, помните про железо, законы вселенной и Халка
- Не стоит пытаться оптимизировать весь код

Литература

- Performance analysis and tuning on modern CPUs, Denis Bakhvalov
- Intel 64 and IA-32 Architectures Optimization Reference Manual
- A Primer on Memory Consistency and Cache Coherence, Second Edition
- [Consistency Models - Jepsen blog](#)
- [Russ Cox's Memory Models series](#)
- [Blog post series on D. Vyukov's 1024cores blog](#)
- [BRAVO – Biased Locking for Reader-Writer Locks](#)
- [D.Vyukov's MPMC bounded queue](#)
- [Designing ASCY-compliant Concurrent Search Data Structures](#) (CLHT)

Ссылка на слайды



Q & A

