

Fast and Scalable Queue-Based Resource Allocation Lock on Shared-Memory Multiprocessors

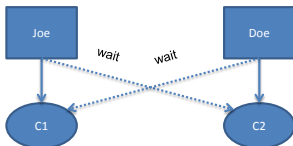
Deli Zhang, Brendan Lynch, and Damian Dechev

University of Central Florida, Orlando, USA

April 27, 2016

Mutual Exclusion on Multicore System

- Mutual exclusion locks are not composable



- Use of multiple mutual exclusion locks poses scalability challenge for data intensive application
 - BerkeleyDB spends over 80% of the execution time in its Test-and-Test-and-Set lock on a 32-core machine¹

¹Johnson et al., Shore-mt: a scalable storage manager for the multicore era. 2009

Resource Allocation Problem

Definition

Given a pool of k resources that require exclusive access, each thread may request $1 \leq h \leq k$ resources, and a thread remains blocked until all required resources are available.

- Resource allocation problem is a generalized mutual exclusion problem (k -mutual exclusion, h -out- k mutual exclusion)
- It is also an extension of the Dining Philosophers Problem (relaxing the static resource configuration)

Locking Protocols

- Assign a mutual exclusion lock to every resource
- Follow protocol to acquire locks one by one

Locking Protocols

- Assign a mutual exclusion lock to every resource
- Follow protocol to acquire locks one by one
 - Two-phase locking

Locking Protocols

- Assign a mutual exclusion lock to every resource
- Follow protocol to acquire locks one by one
 - Two-phase locking
 - Resource hierarchy

Locking Protocols

- Assign a mutual exclusion lock to every resource
- Follow protocol to acquire locks one by one
 - Two-phase locking
 - Resource hierarchy
 - Time-stamp locking

Locking Protocols

- Assign a mutual exclusion lock to every resource
- Follow protocol to acquire locks one by one
 - Two-phase locking
 - Resource hierarchy
 - Time-stamp locking
- Prone to conflict and retry

Batch Locking

- Centralized manager to distribute resources
- It handles resource request from one thread in one batch
 - Extended TATAS
 - Multi-resource lock

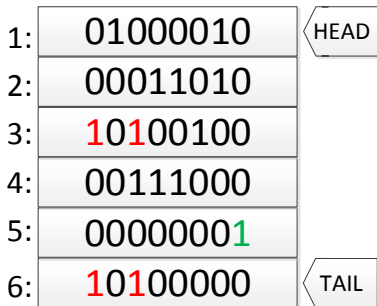
```
typedef uint64 bitset;  
  
void lock(biteset* l, biteset r){  
    biteset b;  
    do{  
        b = *l;  
        if(b & r) //detect conflict  
            continue;  
    }while(CAS(l, b, b | r) != b);  
}
```

- A bitset is an array of bits
- Represent a resource by one bit
- Detecting conflict by bitwise AND
- Handle acquisition in batch

```
typedef uint64 bitset;  
  
void lock(biteset* l, biteset r){  
    biteset b;  
    do{  
        b = *l;  
        if(b & r) //detect conflict  
            continue;  
    }while(CAS(l, b, b | r) != b);  
}
```

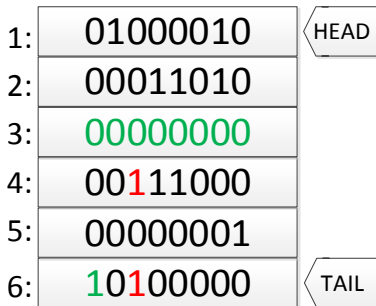
- A bitset is an array of bits
- Represent a resource by one bit
- Detecting conflict by bitwise AND
- Handle acquisition in batch
- Drawbacks
 - No fairness guarantee
 - Heavy contention
 - Limited number of resources

Queue-based Multi-resource Lock



- Ring buffer based concurrent queue
- Resolve conflicts in FIFO order
- Unbounded number of resources

Queue-based Multi-resource Lock



- Ring buffer based concurrent queue
- Resolve conflicts in FIFO order
- Unbounded number of resources

```
struct cell{
    atomic<uint32> seq;
    bitset bits;
}
struct mrlock{
    cell* buffer;
    uint32 siz;
    atomic<uint32> head;
    atomic<uint32> tail;
}
void init(mrlock& l, uint32 siz){
    l.buffer = new cell[siz];
    l.siz = siz;
    l.head.store(0);
    l.tail.store(0);
    for(uint32 i = 0; i < siz; i++){
        l.buffer[i].bits.set();
        l.buffer[i].seq.store(i);
    }
}
```

- Declaration

- *Sequence number* as sentinel
- bits as resource flags
- Atomic queue head and tail

- Initialization

- Allocate adjacent buffer cells
- bits are initialized to 1s
- seq are initialized to cell index

Lock Acquire

```
function ACQUIRE(mrlock* l, bitset r)
  loop
    pos  $\leftarrow$  l.tail, c  $\leftarrow$  ReadCell(l, pos), seq  $\leftarrow$  c.seq
    if seq - pos == 0 then
      if CAS(&l.tail, pos, pos + 1) succeeds then
        break
  c.bits  $\leftarrow$  r, c.seq  $\leftarrow$  pos + 1

  spin_pos  $\leftarrow$  l.head
  while spin_pos != pos do
    if IsDequeued(spin_pos) or NoConflict(spin_pos, r) then
      spin_pos++
  return pos
```

Lock Release

```
function RELEASE(mrlock* l, handle pos)
  ReadCell(l, pos).bits  $\leftarrow$  0
  pos  $\leftarrow$  l.head
  while ReadCell(l, pos).bits == 0 do
    c  $\leftarrow$  ReadCell(l, pos)
    seq  $\leftarrow$  c.seq
    if seq - pos - 1 == 0 then
      if CAS(&l.head, pos, pos + 1) succeeds then
        c.bits  $\leftarrow$  ~0
        c.seq  $\leftarrow$  pos + l.siz
  pos  $\leftarrow$  l.head
```


Sequence Number

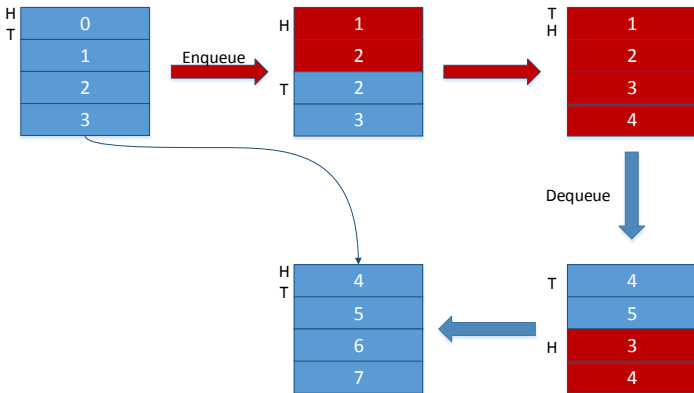


Figure: Updating flow of sequence numbers

Sketch of Correctness Proofs

- Concurrent update of the ring buffer is safe

Theorem

The head always precedes the tail; the tail is larger than head by at most N , where N equals to the size of the buffer.

- Non-atomic update of bitset is safe
 - Bitsets are initialized to 1s
 - Then written to specific request value by one thread

Theorem

*In the presence of a single writer, intermediate values of the bitset during the write operation represent some **supersets** of requested resources.*

- Two-phase locking
 - `std::lock` function with `std::mutex` (STDLock)
 - `boost::lock` function with `boost::mutex` (BSTLock)
- Resource hierarchy
 - with `std::mutex` (RHSTD)
 - with `tbb::queue_mutex` (RHQueue)
- Extended TATAS (ETATAS)

Testing Configurations

- 64-core NUMA system
 - 4 AMD Opteron CPUs with 16 cores per chip @ 2.1GHz
- Micro-benchmark
 - Tight loop to acquire/release locks
 - Randomize resource request prior to the loop
- Configuration
 - Resource: 2 to 1024
 - Threads: 2 to 64
 - Resource contention: 0% to 100% (number of resources requested per thread divided by the total number of resources)

16 Threads

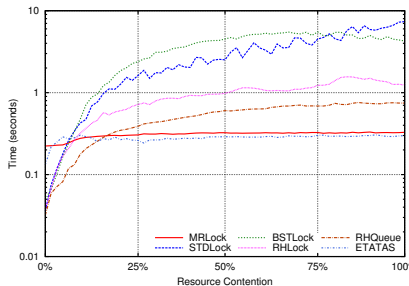


Figure: 64 Resources

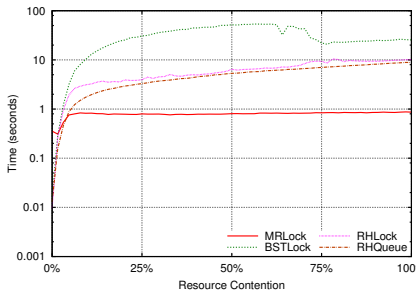


Figure: 1024 Resources

ETATAS, STDLock are excluded on the right because of lack of support for more than 64 resources.

Up to 64 Resources

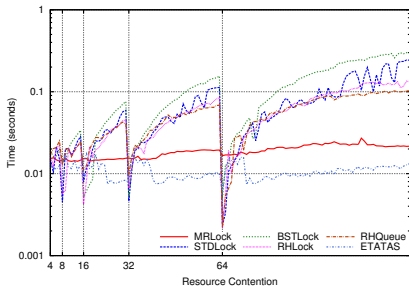


Figure: 2 Threads

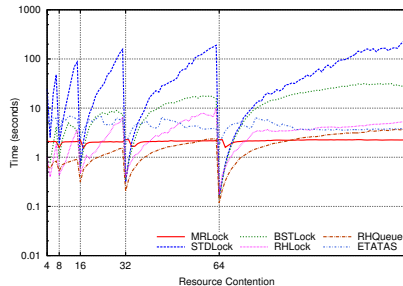


Figure: 64 Threads

Up to 1024 Resources

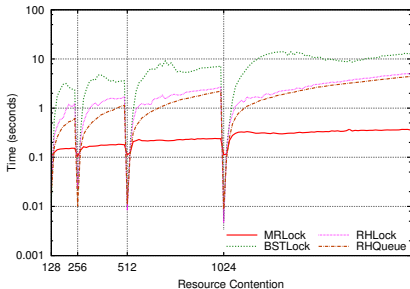


Figure: 8 Threads

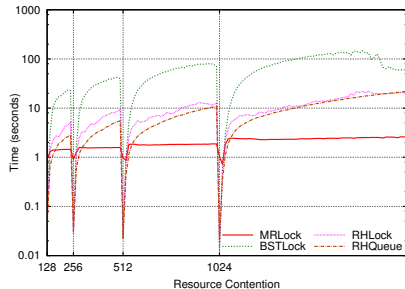


Figure: 32 Threads

Thread Scale

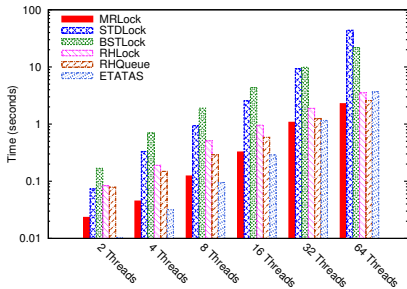


Figure: Contention 32/64 (50%)

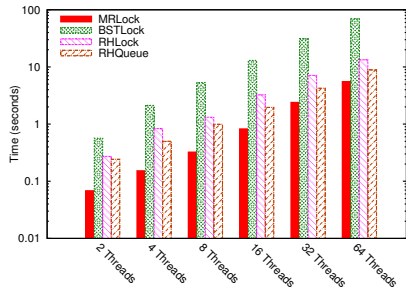


Figure: Contention 128/1024 (12.5%)

Conclusion and Future Work

- Algorithmic Advantage
 - FIFO ordering guarantees fair acquisition of locks
 - Support large number of resources with the use of bitset
 - Performance advantage under mid-to-high levels of contention
- Future Work
 - Adopting wait-free ring buffer to achieve starvation-freedom
 - NUMA-awareness
 - Adapting algorithm to compensate performance under low levels of contention

Questions?

Thank you!