SAND2013-7539C

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

Deli Zhang[1], Brendan Lynch[1], and Damian Dechev[1,2]

1. Department of EECS, University of Central Florida, Orlando, FL 32816, USA
2. Sandia National Laboratories, Livermore, CA 94551, USA
{de-li.zhang, brendan.lynch}@knights.ucf.edu, ddechev@sandia.gov

Abstract. We present a fast and scalable lock algorithm for shared-memory multiprocessors addressing the resource allocation problem, which is also known as the h-out-of-k mutual exclusion problem. In this problem, threads compete for k shared resources where a thread may request an arbitrary number $1 \leq h \leq k$ of resources at the same time. The challenge is for each thread to acquire exclusive access to desired resources while preventing deadlock or starvation. Many existing approaches solve this problem in a distributed system, but the explicit message passing paradigm they adopt is not optimal for shared-memory. Other applicable methods, like two-phase locking and resource hierarchy, suffer from performance degradation under heavy contention, while lacking a desirable fairness guarantee. This work describes the first multi-resource lock algorithm that guarantees the strongest first-in, first-out (FIFO) fairness. Our methodology is based on a non-blocking queue where competing threads spin on previous conflicting resource requests. In our experimental evaluation we compared the overhead and scalability of our lock to the best available alternative approaches using a micro-benchmark. As contention increases, our multi-resource lock obtains an average of ten times speedup over the alternatives including GNU C++'s lock method and Boost's lock function. Additionally, the timings of the multi-resource lock are most consistent in contrast to the existing approaches.

1 Introduction

Improving the scalability of resource allocation algorithms on shared-memory multiprocessors is of practical importance due to the trend of developing many-core chips [7]. The performance of parallel applications on a shared-memory multiprocessor is often limited by contention for shared resources, creating the need for efficient synchronization methods. In particular, the limitations of the synchronization techniques used in existing database systems leads to poor scalability and reduced throughput on modern multicore machines [22]. For example, when running on a machine with 32 hardware threads, Berkeley DB spends over 80% of the execution time in its Test-and-Test-and-Set lock [22].

Mutual exclusion locks eliminate race conditions by limiting concurrency and enforcing sequential access to shared resources. Comparing to more intricate approaches like lock-free synchronization [16] and software transactional memory [20], mutual exclusion locks introduce sequential ordering that eases the reasoning about correctness. Despite the popular use of mutual exclusion locks, one requires extreme caution when using multiple mutual exclusion locks together. In a system with several shared resources, threads often need more than just one resource to complete certain tasks, and assigning one mutual exclusion lock to one resource is common practice. Without coordination between locks this can produce undesirable effects such as deadlock, livelock and decrease in performance.

Consider two clerks, *Joe* and *Doe*, transferring money between two bank accounts C_1 and C_2, where the accounts are exclusive shared resources and the clerks are two contending threads. To prevent conflicting access, a lock is associated with each bank account. The clerks need to acquire both locks before transferring the money. The problem is that mutual exclusion locks cannot be composed, meaning that acquiring multiple locks inappropriately may lead to deadlock. For example, when *Joe* locks the account C_1 then he attempts to lock C_2. In the meantime, *Doe* has acquired the lock on C_2 and waits for the lock on C_1. In general, one seeks to allocate multiple resources among contending threads that guarantees forward system progress, which is known as the resource allocation problem [15]. Two pervasive solutions, namely resource hierarchy [11] and two-phase locking [13], prevent the occurrence of deadlocks but do not respect the fairness among threads and their performance degrades as the level of contention increases. Nevertheless, both the GNU C++ library[1] and the Boost library[2] adopt the two-phase locking mechanism as a means to avoid deadlocks.

In this paper, we propose the first first-in, first-out (FIFO) multi-resource lock algorithm for solving the resource allocation problem on shared-memory multiprocessors. Given k resources, instead of having k separate locks for each one, we employ a non-blocking queue as the centralized manager. Each element in the queue is a resource request bitset[3] of length k with each bit representing the state of one resource. The manager accepts the resource requests in a first-come, first-served fashion: new requests are enqueued to the tail, and then they progress through the queue in a way that no two conflicting requests can reach the head of the queue at the same time. Using the bitset, we detect resource conflict by matching the correspondent bits. The key algorithmic advantages of our approach include:

1. The FIFO nature of the manager guarantees fair acquisition of locks, while implying starvation-freedom and deadlock-freedom
2. The lock manager has low access overhead and is scalable with the cost of enqueue and dequeue being only a single `compare_and_swap` operation

[1] http://gcc.gnu.org
[2] http://www.boost.org
[3] A bitset is a data structure that contains an array of bits.

3. The maximum concurrency is preserved as a thread is blocked only when there are outstanding conflicting resource requests
4. Using a bitset allows an arbitrary number of resources to be tracked with low memory overhead, and does not require atomic access

We evaluate the overhead and scalability of our lock algorithm using a micro-benchmark. We compare our work to the state-of-the-art approaches in the field, which include resource hierarchy locking combined with Intel TBB[4] queue mutex, two-phase locking, such as `std::lock` and `boost::lock`, and an extended Test-and-Test-and-Set (TATAS) lock. At low levels of contention, our lock sacrifices performance for fairness resulting in a worst case slowdown of 2 times. As contention increases, it outperforms the two-phase locking methods by a factor of 10 with a worst case speedup of 1.5 to 2 times against the resource hierarchy lock and the extended TATAS lock. Moreover, the timings of our multi-resource lock are significantly more consistent and regular throughout all test scenarios when compared to other approaches.

The rest of the paper is organized as follows. Section 2 formalizes the resource allocation problem and provides a brief summary on lock-free data structures. Section 4 details the design and implementation of the proposed multi-resource lock algorithm. Section 5 reasons about the safety and progress properties of our algorithm. Section 6 reviews some related works. In section 7 we present the experimental evaluation of our approach. Section 8 concludes with a summary of results and perspectives on future work.

2 Background

In this section, we briefly review the mutual exclusion problem and its variations, with emphasis on the resource allocation problem and the desirable properties for a solution. We also provide a summary on the lock-free data structures and the atomic primitives used in our algorithm.

2.1 Mutual Exclusion and Resource Allocation

Mutual exclusion algorithms are widely used to construct synchronization primitives like locks, semaphores and monitors. Designing efficient and scalable mutual exclusion algorithms has been extensively studied (Raynal [29] and Anderson [1] provide excellent surveys on this topic). In the classic form of the problem, competing threads are required to enter the critical section one at a time. In the k-mutual exclusion problem [15], k units of an identical shared resource exist so that up to k threads are able to acquire the shared resource at once. Further generalization of k-mutual exclusion gives the h-out-of-k mutual exclusion problem [28], in which a set of k identical resources are shared among threads. Each thread may request any number $1 \leq h \leq k$ of the resources, and the thread remains blocked until all the required resources become available.

[4] http://www.threadingbuildingblocks.org

We address the resource allocation problem [24] on shared-memory multiprocessors, which extends the h-out-of-k mutual exclusion problem in the sense that the resources are not necessarily identical. The resource allocation problem can also be seen as a generalization to the prominent *Dining Philosophers Problem* (DPP) originally formulated by Dijkstra [12]. It drops the static resource configuration used in the DPP and allows an arbitrary number of resources to be requested from a pool of k resources. The minimal safety and liveness properties for any solution include mutual exclusion and deadlock-freedom [1]. Mutual exclusion means a resource must not be accessed by more than one thread at the same time, while deadlock-freedom guarantees system wide progress. Starvation-freedom, a stronger liveness property than deadlock-freedom, ensures every thread eventually gets the requested resources. In the strongest FIFO ordering, the threads are served in the order they arrive. It is preferable for ensuring starvation-freedom because it enforces strict fairness between contenders [26]. Another desirable property "local spin" [2] is crucial to the scalability of locks. Local spin algorithms are cache aware so that threads do not make remote memory reference in the spin (busy-wait) loop.

2.2 Atomic Primitives and Synchronization

Atomic primitives are the cornerstones of any synchronization algorithm. `compare_and_swap(address, expectedValue, newValue)`[5], or CAS for short, always returns the original value at the specified `address` but only writes `newValue` to `address` if the original value matches `expectedValue`. CAS is preferred for two reasons: first, it is a *universal* atomic operation (infinite consensus number) [19], thus can be used to construct any other atomic operations; second, it is now widely supported in most systems after first appearing in the IBM 370. A slightly different version `compare_and_set` returns a Boolean value indicating whether the comparison succeeded. In C++ memory model [6], the use of an atomic operation is usually accompanied by `std::memory_order`, which specify how regular memory accesses made by different threads should be ordered around the atomic operation. More specifically, a pair of `std::memory_order_acquire` and `std::memory_order_release` requires that when a thread does a atomic load operation with `acquire` order, prior writes made to other memory locations by the thread that did the `release` become visible to it. `std::memory_order_relaxed`, on the hand, poses no ordering constraints.

Given the atomic CAS instruction, it is straightforward to develop simple spin locks. In Algorithm 1 we present a extended TATAS lock that solves the resource allocation problem for a small number of resources. The basic TATAS lock [30] is a spin lock that allows threads to busy-wait on the initial `test` instruction to reduce bus traffic. The key change we made is to treat the lock integer value as a bit array instead of a Boolean flag. A thread needs to specify the resource requests through a bitset mask when acquiring and releasing the lock. With each bit representing a resource, the bits associated with the desired resources are set

[5] Also known as `compare_exchange`

Algorithm 1 TATAS lock for resource allocation

```
1 typedef uint64 bitset; //use 64bit
      integer as bitset
2
3 //input l: the address of the lock
4 //input r: the resource request bit
      mask
5 void tatas_lock(bitset* l, bitset r
      ){
6   bitset b;
7   do{
8     b = *l; //read bits value
9     if(b & r)    //check for
          confliction
10      continue;  //spin with reads
11   }while(!compare_and_set(l, b, b |
          r));
12 }
13
14 void tatas_unlock(bitset* l, bitset
         r){
15   bitset b;
16   do{
17     b = *l;
18   }while(!compare_and_set(l, b, b &
          ~r));
19 }
```

to 1 while others remain 0. The request updates the relevant bits in the lock bitset if there is no conflict, otherwise the thread spins. One drawback is that the total number of resources is limited by the size of integer type because a bitset capable of representing arbitrary number of resources may span across multiple memory words. Updating multiple words atomically is not possible without resorting to multi-word CAS [17], which is not readily available on all platforms.

Non-blocking synchronization [16], eliminates the use of locks completely. A concurrent object is lock-free if at least one thread makes forward progress in a finite number of steps [18]. It is wait-free if all threads make forward progresses in a finite number of steps. Compared to their blocking counterparts, non-blocking objects promise greater scalability and robustness. One common way to construct a non-blocking object is to use CAS: each contending thread speculates by applying a set of writes on a local copy of the shared data and attempts to CAS the shared object with the updated copy [9]. In this work, we take advantage of a non-blocking queue to increase the scalability and throughput of our lock mechanism.

¡¡¡¡¡¡¡ local

3 Related Work

As noted in section 2.1, a substantial body of work addresses the mutual exclusion problem and the generalized resource allocation problem. In this section, we summarize the solutions to the resource allocation problem that are applicable to shared-memory multiprocessors. We skip the study of the resource allocation problem in distributed environments [4,**?**,**?**]. These solutions do not transfer to shared-memory systems because of the drastically different communication characteristics. In distributed environments processes communicate with each other by message passing, while in shared-memory systems communication is done through shared memory objects. We also omit early mutual exclusion algorithms that use more primitive atomic read and write registers [29,1]. As we show in section 2.2, the powerful CAS operation on modern multiprocessors greatly reduces the complexity of mutual exclusion algorithms.

3.1 Resource Allocation Solutions

Assuming each resource is guarded by a mutual exclusion lock, lock acquiring protocols can effectively prevent deadlocks. Resource hierarchy is one protocol given by Dijkstra [11] based on total ordering of the resources. Every thread locks resources in an increasing order of enumeration; if a needed resource is not available the thread holds the acquired locks and waits. Deadlock is not possible because there is no cycle in the resource dependency graph. Lynch [24] proposes a similar solution based on a partial ordering of the resources. Resource hierarchy is simple and efficient, and has been widely used in practice. However, total ordering requires prior knowledge of all system resources, and dynamically incorporating new resources is difficult. Two-phase locking [13] was originally proposed to address concurrency control in databases. At first, threads are allowed to acquire locks but not release them, and in the second phase threads are allowed to release locks without acquisition. For example, a thread tries to lock all needed resources one at a time; if anyone is not available the thread releases all the acquired locks and start over again. When applied to shared-memory systems, it requires a "try lock" method that returns immediately instead of blocking the thread when the lock is not available. The performance of two-phase locking degrades drastically under contention, because its release-and-wait protocol is vulnerable to failure and retry. Time stamp ordering [5] prevents deadlock by selecting an ordering among the threads. Whenever there is a conflict the thread with smaller time stamp wins. Usually a unique time stamp is assigned to the thread before it starts to lock the resources.

3.2 Queue-Based Algorithms

Fischer et al. [14] describes a simple FIFO queue algorithm for the k-mutual exclusion problem. Awerbuch and Saks [3] proposed the first queuing solution to the resource allocation problem. They treat it as a dynamic job scheduling problem, where each job encapsulates all the resources requested by one process. Newly enqueued jobs progress through the queue if no conflict is detected. Their solution is based on a distributed environment in which the enqueue and dequeue operation are done via message communication. Due to this limitation, they need to assume no two jobs are submitted concurrently. Spin locks such as the TATAS lock shown in Algorithm 1 induce significant contention on large machines, leading to irregular timings. Queue-based spin locks eliminate these problems by making sure that each thread spins on a different memory location [31]. Note that queue-based spin locks do not solve the resource allocation problem directly, but we take inspiration from them when designing our queue-base multi-resource locks. Anderson [2] embeds the queue in a Boolean array. The size of the array equals the number of threads. Each thread determines its unique spin position by drawing a ticket. When relinquishing the lock, the thread resets the Boolean flag on the next slot to notify the waiting thread. The MCS lock [26] designed by Scott et al., employs a linked list with pointers from each thread to its successor. The CLH lock by Craig et al. [8,25] also employs

a linked list but with pointers from each thread to its predecessor. The most notable features of the queue locks are the FIFO ordering of lock acquisitions and the local spin property. =======

4 Algorithms

We implement a queue-based multi-resource lock that manages an arbitrary number of exclusive resources on shared-memory architectures. Our highly scalable algorithm controls resource request conflicts by holding all requests in a FIFO queue and allocating resources to the threads that reach the top of the queue. We achieve scalable behavior by representing resource requests as a bitset and employing a non-blocking queue that grants fair acquisition of the locks. ¿¿¿¿¿¿¿ other

Algorithm 2 Multi-Resource Lock Data Structures

```
1 #include<bitset.h>
2 #include<atomic>
3 using namespace std;
4
5 struct cell{
6   atomic<uint32> seq;  //sequnce
        number
7   bitset bits;         //resource
        request bits
8 }
9 struct mrlock{
10  cell* buffer;        //ring
        buffer of cells
11  uint32 mask;         //mask for
        fast modulation
12  atomic<uint32> head; //head
        position
13  atomic<uint32> tail; //tail
        position
14 }
15
16 //input l: reference to the lock
      instance
17 //input siz: required buffer size (
      power of 2)
18 void init(mrlock& l, uint32 siz){
19   l.buffer = new cell[siz];
20   l.mask = siz - 1;
21   l.head.store(0,
         memory_order_relaxed);
22   l.tail.store(0,
         memory_order_relaxed);
23   //initialize bits to all 1s and
         seq to cell index
24   for (uint32 i = 0; i < siz; i++)
         {
25       l.buffer[i].bits.set();
26       l.buffer[i].seq.store(i,
             memory_order_relaxed);
27   }
28 }
29
30 void uninit(mrlock& l){
31   delete[] l.buffer;
32 }
```

4.1 Bounded Non-Blocking FIFO Queue

Our conflict management approach is built on an array-based bounded lock-free FIFO queue [21]. The lock-free property is desirable as our lock manager must guarantee deadlock freedom. The FIFO property of the data structure allows for serving threads in their arriving order, implying starvation-freedom for all enqueued requests. We favor an array-based queue over other high performance non-blocking queues because it does not require dynamic memory management. Link-list based queues involve dynamic memory allocation for new nodes, which could lead to significant performance overhead and the ABA problem [27]. With

a pre-allocated continuous buffer, our lock algorithm is not prone to the ABA problem and has low runtime overhead by using a single CAS for both enqueue and dequeue operations.

Algorithm 2 defines the lock manager's classes. The `cell` structure defines one element in the queue buffer, it consists of a bitset that represents a resource request and an atomic *sequence number* that coordinates concurrent access. The `mrlock` structure contains a cell buffer pointer, the size mask, and the queue head and tail. We use the size mask to apply fast index modulus. In our implementation, the head and tail increase monotonically; we use an index modulus to map them to the correct array position. Expensive modulo operations can be replaced by bitwise `and` if the buffer size is chosen to be a power of two. We discuss the choice of buffer size in Section 5.2, and explain the initialization of the sequence number and the bitset in following section.

4.2 Acquiring Locks

Given a set of resources, each bit in a request bitset is uniquely mapped to one resource. A thread encapsulates a request of multiple resources in one bitset with the correspondent bit of the requested resources set to `1`. The multi-resource lock handles requests atomically meaning that a request is fulfilled only if all requested resources are made available, otherwise the thread waits in the queue. This all-or-nothing atomic acquisition allows the maximum number of threads, without conflicting requests, to use the shared resources. The length of the bitset is unlimited and can be set either at runtime as in `boost::dynamic_bitset`, or at compile time as in `std::bitset`. Using variable length bitset is also possible to accommodate growing number of total resources at runtime, as long as the resource mapping is maintained.

Figure 3 demonstrates this approach. A newly enqueued request is placed at the tail. Starting from the queue head, it compares the bitset value with each request. In the absence of conflict, it moves on to the next one until it reaches itself. Here, the request is actually conflicting with the fourth request. This causes the tail thread to spin locally on request four until the fourth request is complete. We list the code for `acquire_lock` function in Algorithm 3, which consists of two steps: enqueue and spin.

In Algorithm 3, the code from line 7 to line 16 outlines a CAS-based loop, with threads competing to update the queue tail on line 13. If the CAS attempt succeeds the thread is granted access to the `cell` at the tail position, and the tail is advanced by one. The thread then stores its resource request, which is passed to `acquire_lock` as the variable `r`, in the cell along with a sequence number. The sequence number serves as a sentinel in our implementation. During the enqueue operation the thread assigns a sequence number to its `cell` as it enters the queue as seen on line 18. The nature of a bounded queue allows the head and tail pointers to move through a circular array. Dequeue attempts to increment the head pointer towards the current tail, while a successful call to enqueue will increment the tail pointer pulling it away from head. The sequence numbers are

Algorithm 3 Multi-Resource Lock Acquire

```
1  //input l: referenc to mrlock
       structure
2  //input r: resource request
3  //output : the lock handle
4  uint32 acquire_lock(mrlock& l,
       bitset r){
5    cell* c;
6    uint32 pos;
7    for(;;){      //cas loop, try to
         increase tail
8      pos = l.tail.load(
           memory_order_relaxed);
9      c = &l.buffer[pos & l.mask];
10     uint32 seq = c->seq.load(
           memory_order_acquire);
11     int32 dif = (int32)seq - (int32
           )pos;
12     if(dif == 0){
13       if(l.tail.
             compare_exchange_weak(pos
                 , pos + 1,
                 memory_order_relaxed))
14         break;
15     }
16   }
17   c->bits = r;  // update the cell
         content
18   c->seq.store(pos + 1,
         memory_order_release);
19   uint32 spin = l.head;
20   while(spin != pos){
21     if(pos - l.buffer[spin & l.mask
           ].seq > l.mask || !(l.
           buffer[spin & l.mask].bits
           & r))
22       spin++;
23   }
24   return pos;
25 }
```

initialized on line 26 in Algorithm 2. It is also used by the `release_lock` function in Algorithm 4.

Once a thread successfully enqueues its request, it spins in the while loop on line 20 to 23. It traverses the queue beginning at the head. When there is a conflict of resources indicated by the bitset, the thread will spin locally on the conflicting request. Line 21 displays two conditions that allow the thread to advance: 1) the cell the thread is spinning on is free and recycled, meaning the cell is no longer in front of this thread. This condition is detected by the use of sequence numbers; 2) The request in the cell has no conflict, which is tested by bitwise `and` of the two requests. Once the thread reaches its position in the queue, it is safe to assume the thread has acquired the requested resources. The position of the enqueued request is returned as a handle, which is required when releasing the locks.

4.3 Releasing Locks

The `release_lock` function releases the locks on the requested resources by setting the bitset fields to zero using the lock handle, on line 4 of Algorithm 4. This allows threads waiting for this position to continue traversing the queue. The removal of the request from the queue is delayed until the request in the head cell is cleared (line 6). If a thread is releasing the lock on the head cell, the releasing operation will perform dequeue and recycle the cell. The thread will also examine and dequeue the cells at the top of the queue until a nonzero bitset is found. The code between lines 6 and 17 outlines a CAS loop that is similar to the enqueue function. The difference is that here threads assist each other with the work of advancing the head pointer. With this release mechanism, threads which finish before becoming the head of the queue do not block the other threads.

Algorithm 4 Multi-Resource Lock Release

```
1 //input l: reference to mrlock
      instance
2 //input h: the lock handle
3 void release_lock(mrlock& l, uint32
      h){
4   l.buffer[h & l.mask].bits.reset()
        ;
5   uint32 pos = l.head.load(
        memory_order_relaxed);
6   while(l.buffer[pos & l.mask].bits
        == 0){
7     cell* c = &l.buffer[pos & l.
        mask];
8     uint32 seq = c->seq.load(
        memory_order_acquire);
9     int32 dif = (int32)seq - (int32
        )(pos + 1);
10    if(dif == 0){
11      if(l.head.
          compare_exchange_weak(pos
          , pos + 1,
          memory_order_relaxed)){
12        c->bits.set();
13        c->seq.store(pos + l.mask
            + 1,
          memory_order_release)
          ;
14      }
15    }
16    pos = l.head.load(
        memory_order_relaxed);
17  }
18 }
```

5 Correctness

In this section we reason about the safety and liveness of our algorithm. Our lock manager is safe because it maintains the desired semantics under concurrent acquire and release: all requests to acquire locks are served in FIFO order and a thread must wait until its resource request is not in conflict with previous requests. The underlying lock-free queue guarantees starvation-freedom for threads within the queue and deadlock-freedom for all contending threads.

5.1 Safety

By using the *sequence number* as a sentinel the following properties of our algorithm are guaranteed: 1) The head always precedes the tail, i.e., $H_{pos} \leq T_{pos}$ where H_{pos} and T_{pos} denote the value of head and tail as defined on line 12 and 13 in Algorithm 2. The head advances only if the sequence number of the cell is equal to $H_{pos} + 1$ (line 9 and 10 of Algorithm 4). This occurs when a previous enqueue operation sets the sequence number to $T_{pos} + 1$ (line 18 of Algorithm 3). 2) The tail is at most siz away from the head, i.e., $T_{pos} - H_{pos} \leq siz$ where siz denotes the size of the ring buffer. In other words, the tail cannot overtake the head. The tail advances when the sequence number of the cell is equal to T_{pos} (line 11 and 12 of Algorithm 3). When the tail wraps around the buffer trying to overtake the head, the sequence number of the cell could be either $T_{pos} - siz$ or $T_{pos} - siz + 1$ depending on whether previous enqueue has updated the sequence number. This enqueue will wait until the head advances. Therefore, the cells between head and tail are always valid and store outstanding requests in FIFO order.

The queuing nature of our multi-resource lock allocates a cell exclusively for each contending thread, which drops the limitation of atomic bitset access required by the extended TATAS lock (Algorithm 1). In our algorithm, each bitset is set to 1 for all the bits during initialization (line 25 of Algorithm 2) and then alternates between 0s, 1s (lines 4 and 12 of Algorithm 4), and desired lock values

(line 17 of Algorithm 3). A bitset can have maximum one writer because each cell is allocated to one thread. Regardless the duration of the writing, the bitset maintains its "locking capability" throughout the whole procedure. Since occupied resources are denoted by 1, a bitset of all 1s denotes the set of all resources and any other values denotes a subset of it. Consider updating a bitset from all 1s to any specific request value, it's essentially removing unwanted resources from the set by filling in 0s, thus the intermediate values always represent some supersets of the requested resources. Therefore, it is not possible for any overlapping reading thread to bypass with conflicting request. Similarly, when the bitset is set to all 0s during lock release, the intermediate values always represents some subsets of the requested resources. This prevents the unlocking operation from blocking threads with no conflicting resource request.

5.2 Liveness

Our lock algorithm is deadlock-free for all threads because the concurrent queue we use for our implementation guarantees lock-free progress when it has not reached its capacity. This means that in a scenario of contending threads, at least one thread succeeds when attempting to acquire or release its desired resources. In Algorithm 3 a thread retries its enqueue operation when the CAS update fails (line 13) or the sequence number mismatches (line 12). After loading the most recent value of the tail (line 8), the CAS fails when the tail has been updated by an intervening thread. The sequence number check fails if either the queue is full ($dif < 0$) or the cell has been taken by an intervening thread ($dif > 0$). When an enqueue attempt fails while the queue is not full, this is an indication that another thread must have succeeded in completing an enqueue operation. Therefore, lock-free property is satisfied among all contending threads while starvation-freedom is provided to the threads within the queue. If a wait-free queue is used in place of the lock-free queue, our lock algorithm will provide starvation-freedom for all threads. Such an implementation is possible based on the method proposed by Kogan [23].

If the queue is full, any new enqueue operation waits to insert its request in the queue until some thread relinquishes its locks. For threads with already enqueued requests, a full queue does not impair the correct execution of lock acquisition/release in FIFO order. For threads that is waiting to insert new requests, this may cause performance degeneration and loss of the FIFO fairness guarantee. In practice, we can easily avoid this situation by allocating a sufficiently large buffer. If the number of threads is know beforehand, then a buffer size equal to the thread count will suffice because a thread can only file one request at a time.

6 Related Work

As noted in section 2.1, a substantial body of work addresses the mutual exclusion problem and the generalized resource allocation problem. In this section, we summarize the solutions to the resource allocation problem and related

queue-based algorithms. We skip the approaches targeting distributed environments [4,28]. These solutions do not transfer to shared-memory systems because of the drastically different communication characteristics. In distributed environments processes communicate with each other by message passing, while in shared-memory systems communication is done through shared memory objects. We also omit early mutual exclusion algorithms that use more primitive atomic read and write registers [29,1]. As we show in section 2.2, the powerful CAS operation on modern multiprocessors greatly reduces the complexity of mutual exclusion algorithms.

6.1 Resource Allocation Solutions

Assuming each resource is guarded by a mutual exclusion lock, lock acquiring protocols can effectively prevent deadlocks. Resource hierarchy is one protocol given by Dijkstra [11] based on total ordering of the resources. Every thread locks resources in an increasing order of enumeration; if a needed resource is not available the thread holds the acquired locks and waits. Deadlock is not possible because there is no cycle in the resource dependency graph. Lynch [24] proposes a similar solution based on a partial ordering of the resources. Resource hierarchy is simple to implement, and when combined with queue mutex it is the most efficient existing approach. However, total ordering requires prior knowledge of all system resources, and dynamically incorporating new resources is difficult. Besides, FIFO fairness is not guaranteed because the final acquisition of the resources is always determined by the acquisition last lock in this hold-and-wait scheme. Two-phase locking [13] was originally proposed to address concurrency control in databases. At first, threads are allowed to acquire locks but not release them, and in the second phase threads are allowed to release locks without acquisition. For example, a thread tries to lock all needed resources one at a time; if anyone is not available the thread releases all the acquired locks and start over again. When applied to shared-memory systems, it requires a `try_lock` method that returns immediately instead of blocking the thread when the lock is not available. Two-phase locking is flexible requiring no prior knowledge on resources other than the desired ones, but its performance degrades drastically under contention, because the release-and-wait protocol is vulnerable to failure and retry. Time stamp ordering [5] prevents deadlock by selecting an ordering among the threads. Usually a unique time stamp is assigned to the thread before it starts to lock the resources. Whenever there is a conflict the thread with smaller time stamp wins.

6.2 Queue-Based Algorithms

Fischer et al. [14] describes a simple FIFO queue algorithm for the k-mutual exclusion problem. Awerbuch and Saks [3] proposed the first queuing solution to the resource allocation problem. They treat it as a dynamic job scheduling problem, where each job encapsulates all the resources requested by one process. Newly enqueued jobs progress through the queue if no conflict is detected. Their

solution is based on a distributed environment in which the enqueue and dequeue operation are done via message communication. Due to this limitation, they need to assume no two jobs are submitted concurrently. Spin locks such as the TATAS lock shown in Algorithm 1 induce significant contention on large machines, leading to irregular timings. Queue-based spin locks eliminate these problems by making sure that each thread spins on a different memory location [31]. Anderson [2] embeds the queue in a Boolean array, the size of which equals the number of threads. Each thread determines its unique spin position by drawing a ticket. When relinquishing the lock, the thread resets the Boolean flag on the next slot to notify the waiting thread. The MCS lock [26] designed by Scott et al., employs a linked list with pointers from each thread to its successor. The CLH lock by Craig et al. [8] also employs a linked list but with pointers from each thread to its predecessor. A Recent queue lock based on *flat-combining* synchronization [10] exhibits superior scalability on NUMA architecture than the above classic methods. The flat-combining technique reduce contention by aggregating lock acquisitions in batch and processing them with a combiner thread. A key difference between this technique and our multi-resource lock is that our method aggregates lock acquisition requests for multiple resources from one thread, while the flat-combining lock gathers requests from multiple threads for one resource. Although the above queue-based locks could not solve the resource allocation problem on their own, they share the same inspiration with our method: using queue to reduce contention and provide FIFO fairness.

7 Performance Evaluation

In this section, we assess the overhead, scalability and performance consistency of our multi-resource lock (MRLock) and compare it with the `std::lock` function from GCC 4.7 (STDLock), the `boost::lock` function from Boost library 1.49 (BSTLock), the resource hierarchy scheme combined with both `std::mutex` (RHSTD) and `tbb::queue_mutex` from Intel TBB library 4.1 (RHQueue), and the extended TATAS lock (ETATAS) mentioned in Algorithm 1. Both `std::lock` and `boost::lock` implement a two-phase locking scheme to acquire multiple locks at the same time without deadlock, but they have slightly different requirements for input parameters: the boost version accepts a range of lock iterators; the standard library version, which takes advantage of the C++ 11 variadic templates, accepts a variable number of locks as function arguments. We use `std::mutex` as the underlying lockable object for `std::lock`, and use `boost::mutex` for `boost::lock`. For resource hierarchy scheme, we choose `std::mutex` as a baseline, and choose `tbb::queue_mutex` as the representative queue lock implementation because it is highly optimized for x86 architecture and readily available.

The evaluation experiments employ a micro-benchmark to simulate the resource allocation problem. It consists of a tight loop that acquires and releases a predetermined number of locks. The loop increments a set of counters, which are simply integers representing the resources. The counters are not atomic, so

without locking their value will be incorrect due to data races. We check the counter against data races to verify the correctness of our lock implementations.

Algorithm 5 lists the benchmark function and related parameters.

Algorithm 5 Micro-Benchmark

```
 1: function Main()
 2:     threads = CreateThreads(TestLock, n)
 3:     WaitForBarrier()
 4:     BeginTimer()
 5:     WaitForThreads(threads)
 6:     EndTimer()

 7: function TestLock(lock, resource, contention, iteration)
 8:     requested = Random(resources, contention)
 9:     WaitForBarrier()
10:     for 1 → iteration do
11:         lock.Acquire(requested)
12:         requested.IncreaseCount()
13:         lock.Release(requested)
```

All tests are conducted on a 64-core ThinkMate RAX QS5-4410 server running Ubuntu 12.04 LTS. It is a NUMA system with four AMD Opteron 6272 CPUs (16 cores per chip @2.1 GHz) and 64 GB of shared memory (16 × 4GB PC3-12800 DIMM). Both the micro-benchmark and the lock implementations are compiled with GCC 4.7 (with the options `-std=c++0x` to enable C++ 11 support).

When evaluating classic mutual exclusion locks, one may increase the number of concurrent threads to investigate their scalability. Since all threads contend for a single critical section, the contention level scales linearly with the number of threads. However, the amount of contention in the resource allocation problem can be raised by either increasing the number of threads or the size of resource request per thread. Given k total resources with each thread requesting h of them, we denote the *resource contention* by the fraction h/k or its quotient in percentage. This notation reveals that *resource contention* may be comparable even though the total number of resources is different. For example, 8/64 or 12.5% means each request needs 8 resources out of 64, which produces about the same amount of contention as 4/32. We show more results in Section 7.2. The product of the thread number p and *resource contention* level roughly represents the overall contention level.

To fully understand the efficiency and scalability in these two dimensions, we test the locks in a wide range of parameter combinations: for thread number $2 \leq p \leq 64$ and for resource number $4 \leq k \leq 64$ each thread requests the same number of resources $2 \leq h \leq k$. We set the loop iteration in the micro-benchmark to 10,000 and get the average time out of 10 runs for each configuration.

7.1 Single-thread Overhead

To measure the lock overhead in the absence of contention, we run the micro-benchmark with a single thread requesting two resources and subtract the loop overhead from the results. Table 1 shows the total timing for the completion of a pair of lock and unlock operations. In this scenario MRLock is slightly slower than ETATAS because of the extra queue traversing operation. The other four methods take about twice the time of MRLock. Each other method takes at a minimum two lock operations, to be considered a solution to the resource allocation problem. As a baseline performance metric we compare against a single `std::mutex` as shown in Table 1.

Table 1. Lock overhead obtained without contention

Lock Mechanism	Overhead
MRLock	$42ns$
STDLock	$95ns$
BSTLock	$105ns$
RHLock	$88ns$
RHQueue	$90ns$
ETATAS	$34ns$
`std::mutex` *	$35ns$
`boost::mutex` *	$35ns$

* overhead of `std::mutex` and `boost::mutex` is provided for reference only since it does not solve resource allocation problem.

7.2 Resource Scalability

Our performance evaluation exploring the scalability of the tested approaches when increasing the level of *resource contention* is shown in Figures 4, 5 and 6. The y axis represents the total time needed to complete the micro-benchmark in a logarithmic scale, where a smaller value indicates better performance. The x axis represents levels of resource contention. For example, the section between 32 and 64 has a total of 32 resources, while the section to the right of 64 has 64 resources. Within each section, the level of contention increases from 1% to 100%. We observe a saw pattern because the resource contention level alternates as we move along the x axis. In addition, we observe that the timing pattern is similar among different sections, supporting our argument that the contention is proportional to the quotient of the request size divided by total number of resources. We also show a zoomed-in view of a single section in Figure 7, which illustrates the timings of 16 threads contending for 64 resources.

When increasing the number of requested resources per thread, the probability of threads requesting the same resources increases. This poses scalability

challenges for both two-phase locks and the RHLock because they rely on a certain protocol to acquire the requested locks one by one. As the request size scales up, the acquiring protocol is prolonged thus prone to failure and retry. Especially in the case of 64 threads (Figure 6), std::lock is more than 50 times slower when the level contention exceeds 75%. Boost::lock exhibits the same problem, it closely resembles std::lock. Unlike the above two methods, RHLock acquires locks in a fixed order, and it does not release current locks if a required resource is not available. This hold-and-wait paradigm helps to stabilize the timings and to reduce the overall contention. RHLock resembles the performance of std::lock in the two thread scenario (Figure 4), but it outperforms both boost::lock and std::lock by about three times under 50% resource contention on 16 threads (Figure 7).

While the time of all the other methods show linear growth with respect to resource contention, MRLock remains constant through out all scenarios. In the case of 64 threads and the request size of 32, MRLock achieves a 20 times speed up over the std::lock, 10 times over the boost::lock and 2.5 times over the RHLock. The fact that MRLock provides a centralized manager to respond the lock requests from threads in one CAS contributes to this high degree of scalability. TATAS lock also adopts the same all-or-nothing scheme, it could be seen as a MRLock algorithm with a queue size of one. It outperforms MRLock on two threads by about 40% (Figure 4), and almost ties with MRLock on 32 threads. However, MRLock is 1.7 time faster on 64 threads, because the queuing mechanism relieves the contention of the CAS loop.

In Figure 7, we see that under low contention levels (less than 10% or 7/64), MRLock and TATAS lock no longer hold an advantage over the RHLock, and the two-phase locks. For example the std::lock only takes 0.003s under contention 2/64 on 64 threads, while MRLock takes 0.016s. When the resource contention level is low, locking protocols are not likely to encounter any conflicts. Since the locks are distributed, the lock acquiring overhead is close to a single thread scenario. This is not the case for MRLock. In the presence of the lock manager, every thread has to enqueue its requests. The queue becomes a single point of contention.

7.3 Thread Scalability

Figures 9, 10 and 11 show the benchmark time of threads under the contention level of 4/64, 8/64 and 32/64 respectively. In these graphs, the contention level is fixed and we investigate the performance scaling characteristics by increasing the number threads. We cluster five approaches on the x axis by the number of threads, while the y axis represents the benchmark time.

When the level of contention is low, MRLock and TATAS lock do not have performance advantages other the other approaches. This is shown in Figure 9. The worst case slowdown happens on 32 threads, where MRLock is 3.7 times slower than std::lock. The difference decreases to about 2 times on 64 threads, which implies that our approach has a smaller scaling factor. We also observe better scalability against TATAS lock, when moving from 32 threads to 64 threads

the performance of TATAS lock degrades three folds resulting a 2 times slow down compared to MRLock.

The contention level that is a turning point for our algorithm's performance is about 12.5% as shown in Figure 10. MRLock ties with RHLock and outperforms all the others. It is 4 times faster than std::lock and twice as fast as TATAS on 64 threads, and also exhibits better scalability. The time of TATAS, boost::lock and std::lock almost tripled when the number of threads is increased from 32 to 64, while the time of MRLock only increases by 100%

In Figure 11, we use a logarithmic scale on the y axis because std::lock takes more than 20 times longer than MRLock, dwarfing the other methods in a linear scale. MRLock outperforms all other methods on all scales except for the TATAS lock. While the TATAS lock has a slight advantage on small thread counts, MRLock catches up by 16 threads, and finally overtakes TATAS on 32 threads. We also observe similar results in Figures 12.

Overall, MRLock exhibits good scalability on all levels of contention, it outperforms std::lock and boost::lock by 10 to 20 times in regions of high contention levels. It is also faster than the RHLock by a factor of 1.5 to 2.5. Even though it does not hold an advantage against the TATAS lock when thread counts are low, it outperforms the TATAS lock by at least 2 times on 64 threads.

7.4 Performance Consistency

It is often desirable that an algorithm produces predicable execution times, especially in real time systems. We demonstrate in Section 7.2 that our multi-resource lock obtains reliable run times regardless the level of resource contention. We illustrate further that our lock implementation achieves more consistent timings among different runs than competing implementations.

Figures 14 and 15 display the standard deviation of execution times from 10 different runs. We generate randomized resource requests at the beginning of each test run (Algorithm 5), so the actual resource conflicts might be different for each run. We show the absolute value of the standard deviation on the y axis, and the number of threads on the x axis (Both are in logarithm scale). Overall, the deviation of all approaches grows slightly as the number of threads increases. This is expected because in regions of high parallelism, the operating system itself contributes a large part of this noise. By looking at the absolute values, MRLock achieves the lowest variation, which means it also outperforms TATAS in terms of consistency. This indicates that the incorporation of a FIFO queue stabilized our lock algorithm. Notably, the deviation of std::lock grows linearly. We also include the percentage deviation in Figures 17 and 16. The y axis gives the percentage error normalized by the average time, the variation of MRLock is within 2% or its executing time.

8 Conclusion and Future Work

Our multi-resource lock algorithm (MRLock) provides a robust solution to the resource allocation problem on shared-memory multiprocessors. The MRLock

algorithm is lightweight and scalable so the overhead of our algorithm has a minimal increase over other practical solutions. Other solutions might be practical in systems with low contention. As shown in the 7 section our solutions reliability and scalability should be preferred in systems with high contention or when system scalability is desired. At all levels of contention our algorithm provides a fairness guarantee that could be desirable in certain use cases even under minimal contention.

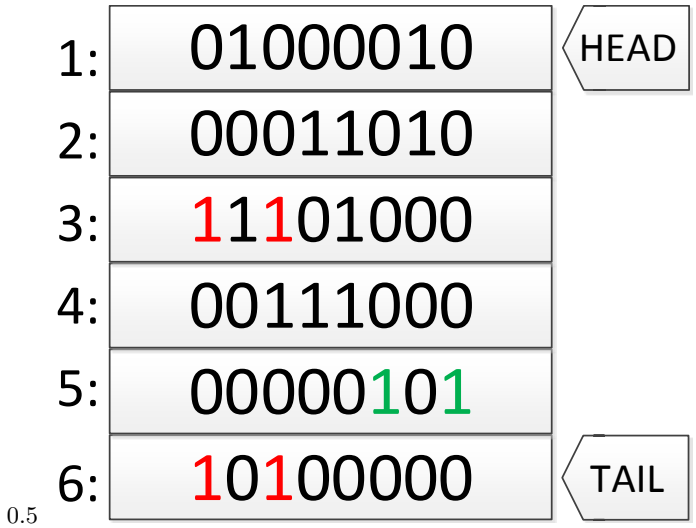
Possible extension for this algorithm includes creating an adaptive method to choose from several locking algorithms based on the level of contention in a system. This would further increase the scalability of the algorithm as it would guarantee the best performance from contention levels of zero to one hundred percent.

Acknowledgment Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04- 94AL85000.

References

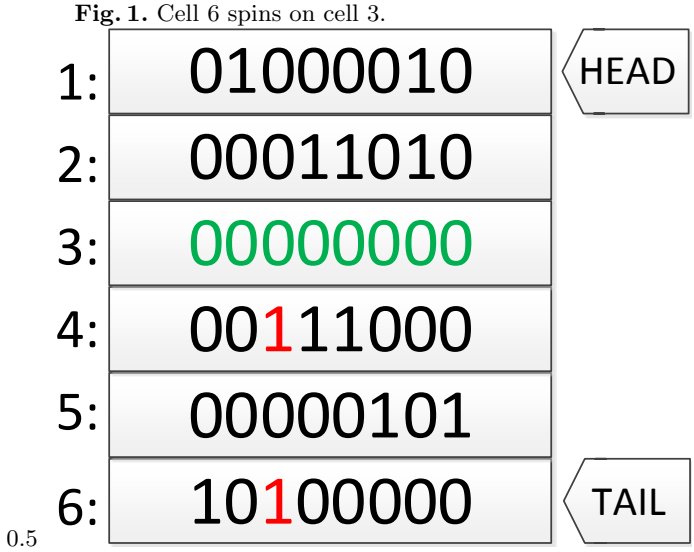
1. Anderson, J., Kim, Y., Herman, T.: Shared-memory mutual exclusion: Major research trends since 1986. Distributed Computing 16(2), 75–110 (2003)
2. Anderson, T.E.: The performance of spin lock alternatives for shared-money multiprocessors. Parallel and Distributed Systems, IEEE Transactions on 1(1), 6–16 (1990)
3. Awerbuch, B., Saks, M.: A dining philosophers algorithm with polynomial response time. In: Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on. pp. 65–74. IEEE (1990)
4. Bar-Ilan, J., Peleg, D.: Distributed resource allocation algorithms. In: Distributed Algorithms, pp. 277–291. Springer (1992)
5. Bernstein, P., Goodman, N.: Timestamp based algorithms for concurrency control in distributed database systems. In: Proceedings 6th International Conference on Very Large Data Bases (1980)
6. Boehm, H.J., Adve, S.V.: Foundations of the c++ concurrency memory model. In: ACM SIGPLAN Notices. vol. 43, pp. 68–78. ACM (2008)
7. Borkar, S.: Thousand core chips: a technology perspective. In: Proceedings of the 44th annual Design Automation Conference. pp. 746–749. ACM (2007)
8. Craig, T.: Building fifo and priorityqueuing spin locks from atomic swap. Tech. rep., Citeseer (1994)
9. Dechev, D., Pirkelbauer, P., Stroustrup, B.: Lock-free dynamically resizable arrays. In: Principles of Distributed Systems, pp. 142–156. Springer (2006)
10. Dice, D., Marathe, V.J., Shavit, N.: Flat-combining numa locks. In: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures. pp. 65–74. ACM (2011)
11. Dijkstra, E.: Hierarchical ordering of sequential processes. Acta informatica 1(2), 115–138 (1971)

12. Dijxstra, E.: Solution of a problem in concurrent programming control. Facultad de Ingenierías Fundaci ó n Universitaria Luisamig ó p. 50 (1965)
13. Eswaran, K., Gray, J., Lorie, R., Traiger, I.: The notions of consistency and predicate locks in a database system. Communications of the ACM 19(11), 624–633 (1976)
14. Fischer, M.J., Lynch, N.A., Burns, J.E., Borodin, A.: Distributed fifo allocation of identical resources using small shared space. ACM Transactions on Programming Languages and Systems (TOPLAS) 11(1), 90–114 (1989)
15. Fischer, M., Lynch, N., Burns, J., Borodin, A.: Resource allocation with immunity to limited process failure. In: Foundations of Computer Science, 1979., 20th Annual Symposium on. pp. 234–254. IEEE (1979)
16. Fraser, K., Harris, T.: Concurrent programming without locks. ACM Transactions on Computer Systems (TOCS) 25(2), 5 (2007)
17. Harris, T.L., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In: Distributed Computing, pp. 265–279. Springer (2002)
18. Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Transactions on Programming Languages and Systems (TOPLAS) 15(5), 745–770 (1993)
19. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(1), 124–149 (1991)
20. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21(2), 289–300 (1993)
21. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann (2012)
22. Johnson, R., Pandis, I., Hardavellas, N., Ailamaki, A., Falsafi, B.: Shore-mt: a scalable storage manager for the multicore era. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology. pp. 24–35. ACM (2009)
23. Kogan, A., Petrank, E.: A methodology for creating fast wait-free data structures. In: ACM SIGPLAN Notices. vol. 47, pp. 141–150. ACM (2012)
24. Lynch, N.: Fast allocation of nearby resources in a distributed system. In: Proceedings of the twelfth annual ACM symposium on Theory of computing. pp. 70–81. ACM (1980)
25. Magnusson, P., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. In: Parallel Processing Symposium, 1994. Proceedings., Eighth International. pp. 165–171. IEEE (1994)
26. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems (TOCS) 9(1), 21–65 (1991)
27. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. pp. 267–275. ACM (1996)
28. Raynal, M.: A distributed solution to the k-out of-m resources allocation problem. Advances in Computing and Information-ICCI'91 pp. 599–609 (1991)
29. Raynal, M., Beeson, D.: Algorithms for mutual exclusion. MIT Press (1986)
30. Rudolph, L., Segall, Z.: Dynamic decentralized cache schemes for mimd parallel processors. In: Proceedings of the 11th annual international symposium on Computer architecture. pp. 340–347. ISCA '84, ACM (1984)
31. Scott, M.L., Scherer, W.N.: Scalable queue-based spin locks with timeout. In: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming. pp. 44–52. PPoPP '01, ACM (2001)



0.5

Fig. 1. Cell 6 spins on cell 3.



0.5

Fig. 2. Release of cell 3. Cell 6 spins on cell 4

Fig. 3. Atomic lock acquisition process

0.5

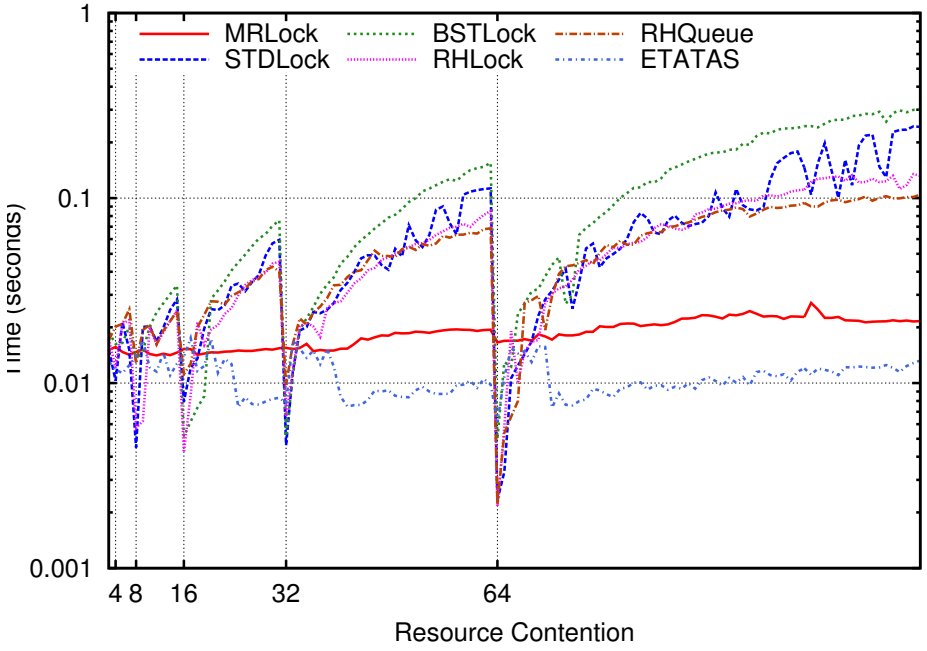


Fig. 4. 2 threads

0.5

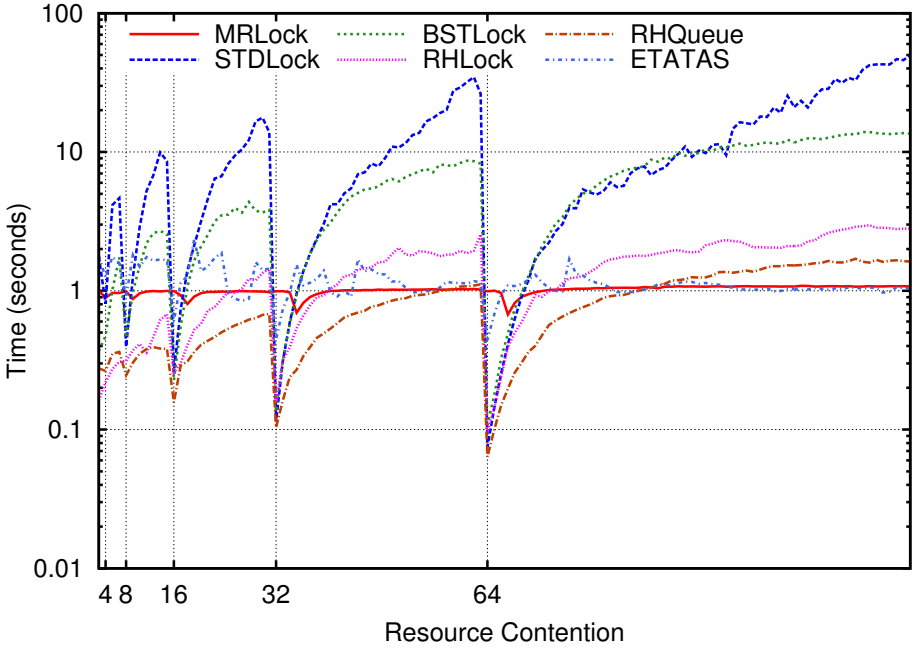
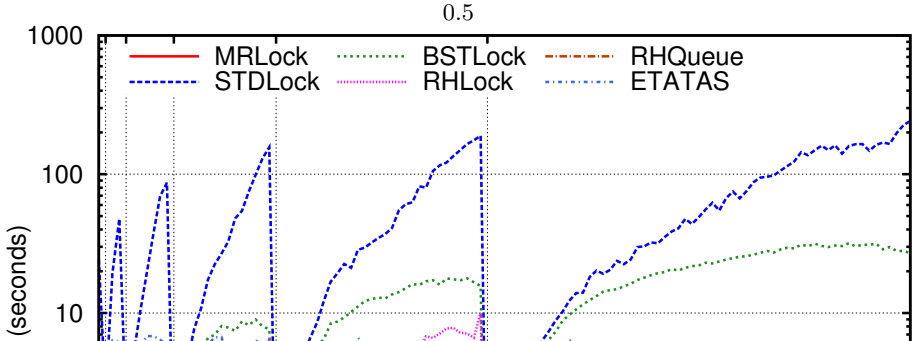


Fig. 5. 32 threads

0.5



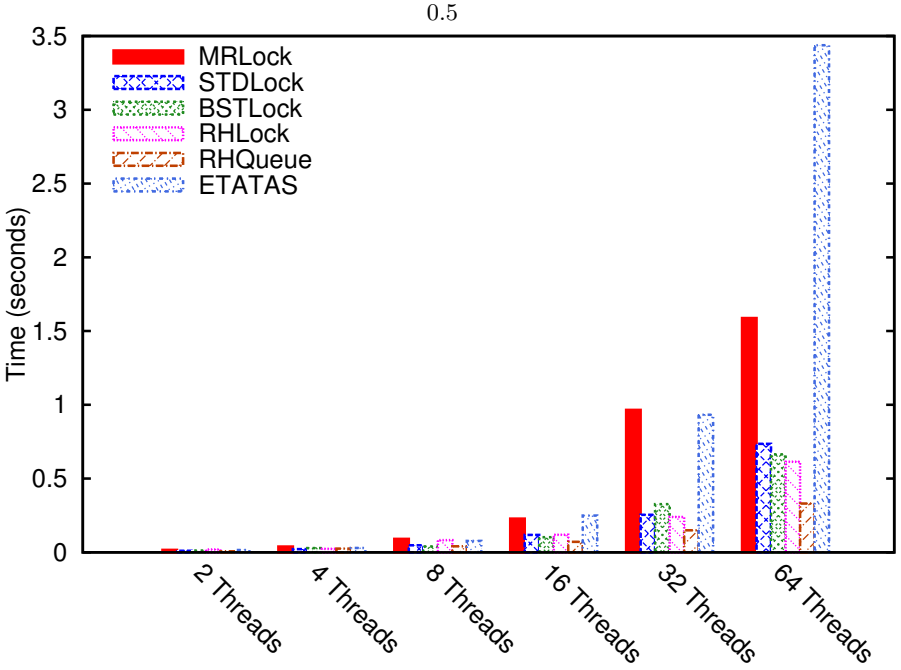


Fig. 9. resource contention 4/64

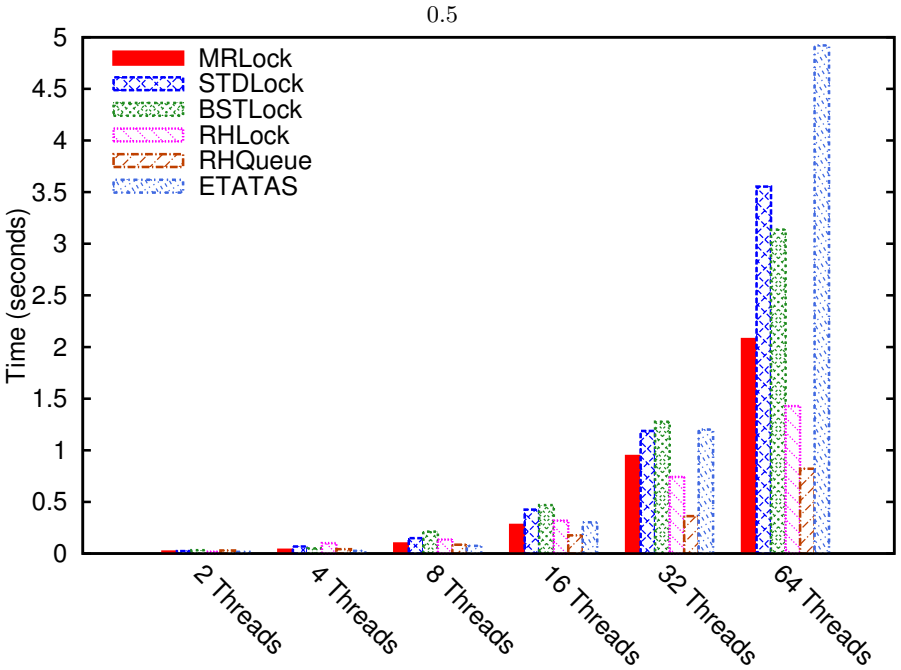
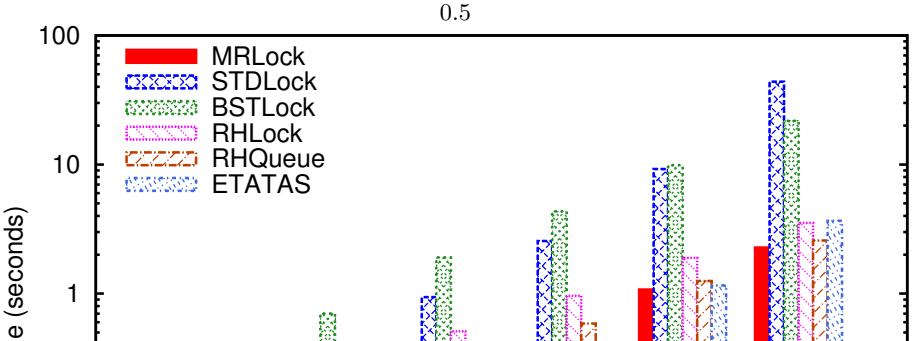


Fig. 10. resource contention 8/64



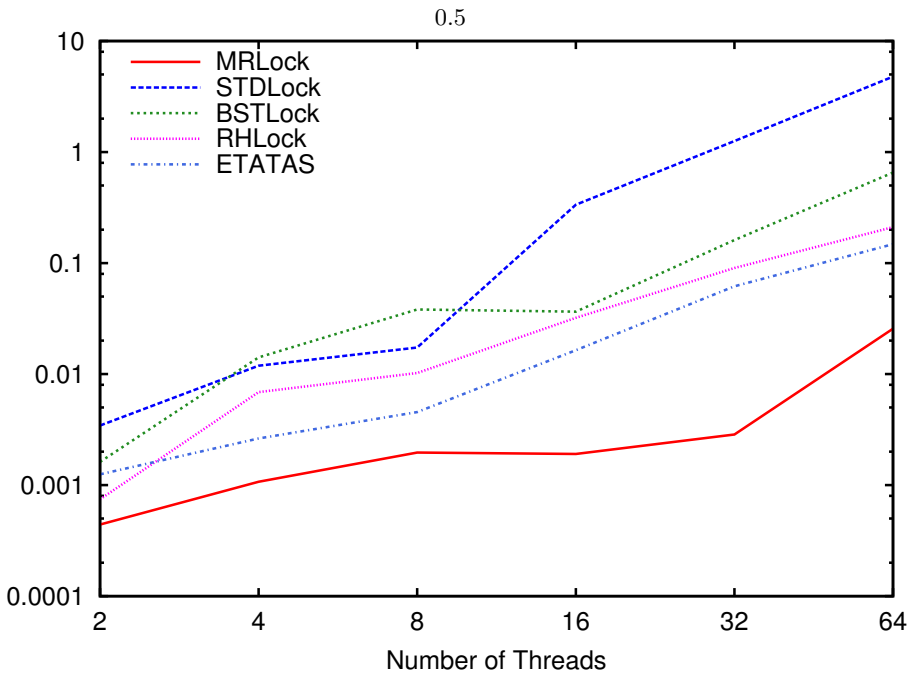


Fig. 14. Resource contention 16/32

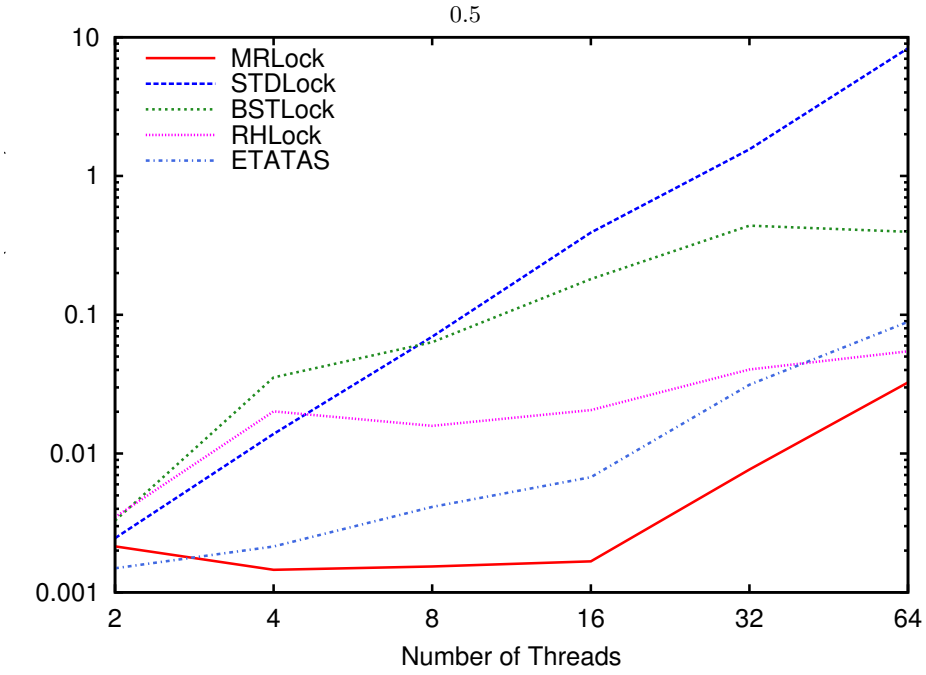


Fig. 15. Resource contention 32/64