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

Deli Zhang, Brendan Lynch, and Damian Dechev

Department of EECS, University of Central Florida
Orlando, FL 32816, USA
{de-li.zhang,brendan.lynch}@knights.ucf.edu, dechev@eecs.ucf.edu

Abstract. We present a fast and scalable lock algorithm for shared-memory multiprocessors addressing the resource allocation 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.

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. 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 [16]. 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 [16].

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 [15] and software transactional memory [14], mutual exclusion locks introduce sequential ordering that eases the

R. Baldoni, N. Nisse, and M. van Steen (Eds.): OPODIS 2013, LNCS 8304, pp. 266–280, 2013.

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 [12]. Two pervasive solutions, namely resource hierarchy [9] and two-phase locking [10], 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 FIFO (first-in, first-out) 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
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

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

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 `std::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.

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 [21] 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 [12], 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 [20], 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.

We address the resource allocation problem [17] 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 [9]. 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 [18].

2.2 Atomic Primitives and Synchronization

Atomic primitives are the cornerstones of any synchronization algorithm. `compare_and_swap(address, expectedValue, newValue)`[4], 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`. A slightly different version `compare_and_set` returns a Boolean value indicating whether the comparison succeeded. In C++ memory model, the use of an atomic operation is 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.

Algorithm 1. TATAS lock for resource allocation

```
1  typedef uint64 bitset;
2
3  //input l: address of the lock
4  //input r: request bit mask
5  void 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 unlock(bitset* l, bitset r){
15   bitset b;
16   do{
17     b = *l;
18   }while(!compare_and_set(l, b, b &
           ~r));
19 }
```

Given the atomic CAS instruction, it is straightforward to develop simple spin locks. In Algorithm 1 we present an extended TATAS lock that solves the resource allocation problem for a small number of resources. The basic TATAS lock 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 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 of this extension 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 [13], which is not readily available on all platforms.

Non-blocking synchronization, 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 [15]. It is wait-free if all threads make forward progresses

[4] Also known as `compare_exchange`

in a finite number of steps [7]. Compared to their blocking counterparts, non-blocking objects promise greater scalability and robustness. In this work, we take advantage of a non-blocking queue to increase the scalability and throughput of our lock mechanism.

3 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.

3.1 Handle Locking Request with FIFO Queue

Our conflict management approach is built on an array-based bounded lock-free FIFO queue [15]. 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 [19]. 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.

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.

(a) Cell 6 spins on cell 3

(b) Release of cell 3. Cell 6 spins on cell 4

Fig. 1. Atomic lock acquisition process

The length of the bitset is unlimited and can be determined 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 1a 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 thread on 5th cell successfully acquires all needed resources. The thread on the tail (6th cell) spins on the 3rd request due to conflict. In Figure 1b, the thread on the tail proceeds to spin on the 4th cell when the 3rd request was released.

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;
7    bitset bits;
8  }
9  struct mrlock{
10   cell* buffer;
11   uint32 mask;
12   atomic<uint32> head;
13   atomic<uint32> tail;
14 }
15
16 //input l: reference to the lock
17 //input siz: suggested buffer size
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
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 }
```

Algorithm 2 defines the lock manager's class. The cell structure defines one element in the queue, 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 operation can be replaced by bitwise AND operation if the buffer size is chosen to be a power of two.

3.2 Acquiring and Releasing Locks

We list the code for lock acquire function in Algorithm 3, which consists of two steps: enqueue and spin. The code from line 7 to 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 lock function 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

Algorithm 3. Lock Acquire

```
1  //input l: referenc to mrlock
2  //input r: resource request
3  //output : the lock handle
4  uint32 lock(mrlock& l, bitset r){
5    cell* c;
6    uint32 pos;
7    for(;;){
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;
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 }
```

Algorithm 4. Lock Release

```
1  //input l: reference to mrlock
2  //input h: the lock handle
3  void unlock(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 }
```

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 initialized on line 26 in Algorithm 2.

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.

The `unlock` 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.

4 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 [4,20]. 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 [21,1]. As we show in section 2.2, the powerful CAS operation on modern multiprocessors greatly reduces the complexity of mutual exclusion algorithms.

4.1 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 [9] 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 [17] 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 [10] 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.

4.2 Queue-Based Algorithms

Fischer et al. [11] 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 [22]. 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 [18] designed by Scott et al., employs a linked list with pointers from each thread to its successor. The CLH lock by Craig et al. [6] also employs a linked list but with pointers from each thread to its predecessor. A Recent queue lock based on *flat-combining* synchronization [8] 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.

5 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 `std::mutex` (RHSTD), and the extended TATAS lock (ETATAS) described in Algorithm 1. We use

`std::mutex` as the underlying lockable object for `std::lock`, and `boost::mutex` for `boost::lock`. These alternative approaches have been widely used in practice and highly optimized for our testing system.

We employ a micro-benchmark to evaluate the performance of these approaches for multiple resource allocation. It consists of a tight loop that acquires and releases a predetermined number of locks. The loop increments a set of integer counters, where each counter represents a resource. The counters are not atomic, so without the use locks their value will be incorrect due to data races. When the micro-benchmark's execution is complete, we check each counter's value to verify the absence of data races and validate the correctness of our lock implementations. 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 ($16 \times$ 4GB PC3-12800 DIMM). Both the micro-benchmark and the lock implementations are compiled with GCC 4.7 (with the options `-o1 -std=c++0x` to enable level 1 optimization and 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 the 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 benchmark timing results in Section 5.2 that verifies this hypothesis. 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.

5.1 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 because each of them takes at a minimum two lock operations to solve a non-trivial resource allocation problem. Although `std::mutex` and `boost::mutex` does not solve the resource allocation problem, we compare against them as a baseline performance metric.

Table 1. Lock overhead obtained without contention

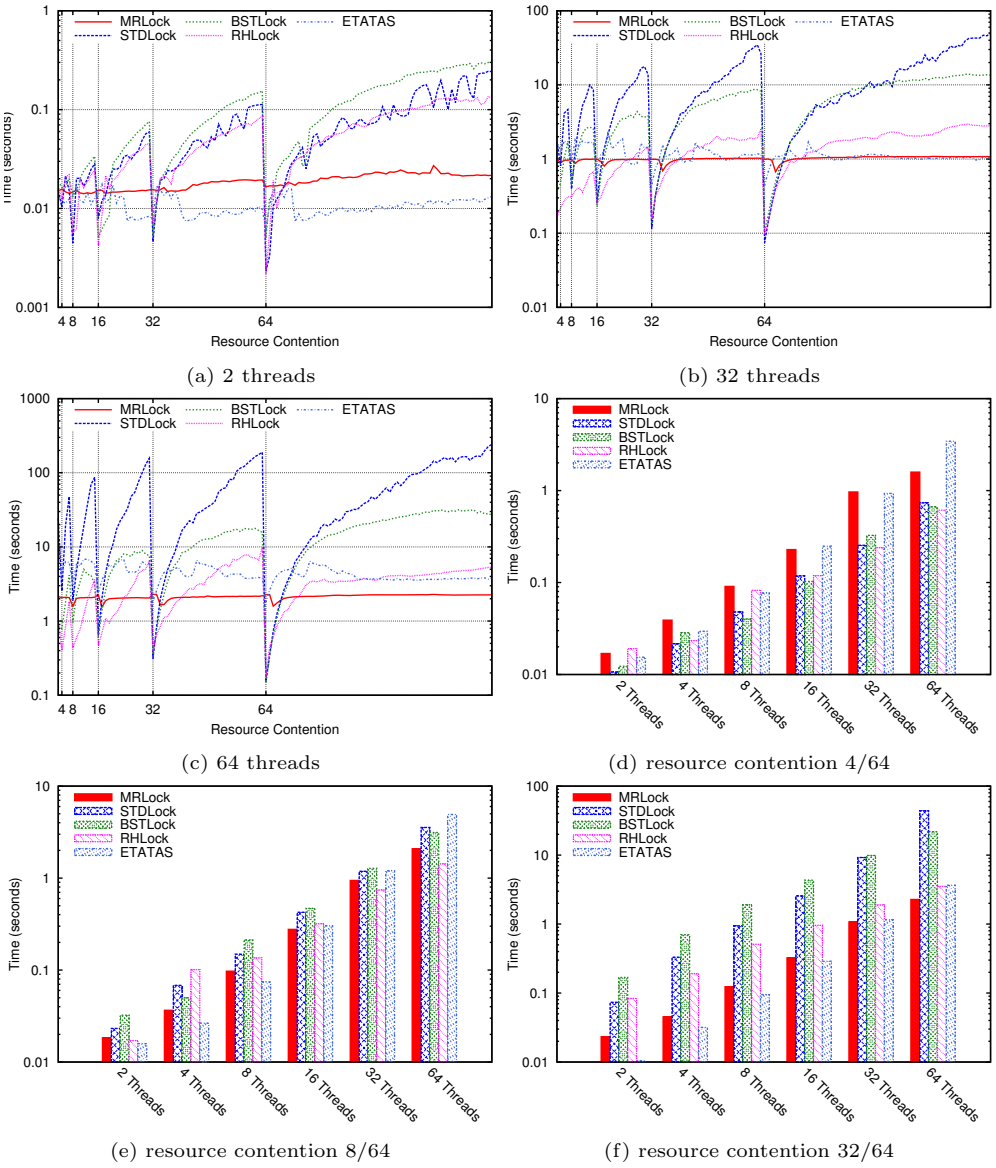
MRLock	STDLock	BSTLock	RHLock	ETATAS	`std::mutex`	`boost::mutex`
$42ns$	$95ns$	$105ns$	$88ns$	$34ns$	$35ns$	$35ns$

5.2 Resource Scalability

Our performance evaluation exploring the scalability of the tested approaches when increasing the level of *resource contention* is shown in Figures 2a, 2b and 2c. 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, and it is divided by five tick marks into six sections. Each tick mark on the x-axis represents the beginning of the section to its right, and the tick mark label denotes the total number of resources in that section. For example, the section between Tick 32 and 64 has a total of 32 resources, while the section to the right of Tick 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.

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 resource hierarchy implementations because they rely on a certain protocol to acquire the requested locks one by one. As the request size increases, the acquiring protocol is prolonged thus prone to failure and retry. At high levels of contention, such as the case with 64 threads (Figure 2c) when the level contention exceeds 75%, STDLock is more than 50 times slower when compared to MRLock. BSTLock exhibits the same problem, and its observed performance closely resembles STDLock's performance. 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 stabilize the timings and reduce the overall contention. RHLock resembles the performance of STDLock in the two thread scenario (Figure 2a), but it outperforms both BSTLock and STDLock by about three times under 50% resource contention on 16 threads (Figure 2b).

While the time of all alternative methods show linear growth with respect to resource contention, MRLock remains constant throughout all scenarios. In the case of 64 threads and request size of 32, MRLock achieves a 20 times speedup over STDLock, 10 times performance gain over BSTLock and 2.5 times performance increase over RHLock. The fact that MRLock provides a centralized manager to respond the lock requests from threads in one batch contributes to this high degree of scalability. ETATAS also adopts the same all-or-nothing scheme, thus it could be seen as an MRLock algorithm with a queue size of



(a) 2 threads

(b) 32 threads

(c) 64 threads

(d) resource contention 4/64

(e) resource contention 8/64

(f) resource contention 32/64

Fig. 2. Performance scaling when increasing resource contention (2a, 2a, and 2a) and the number of threads (2d, 2e, and 2f)

one. It outperforms MRLock on two threads by about 40% (Figure 2a), 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.

5.3 Thread Scalability

Figures 2d, 2e and 2f show the execution time for our benchmark in the scenarios when the threads experience contention levels 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 total time needed to complete the benchmark in logarithm scale.

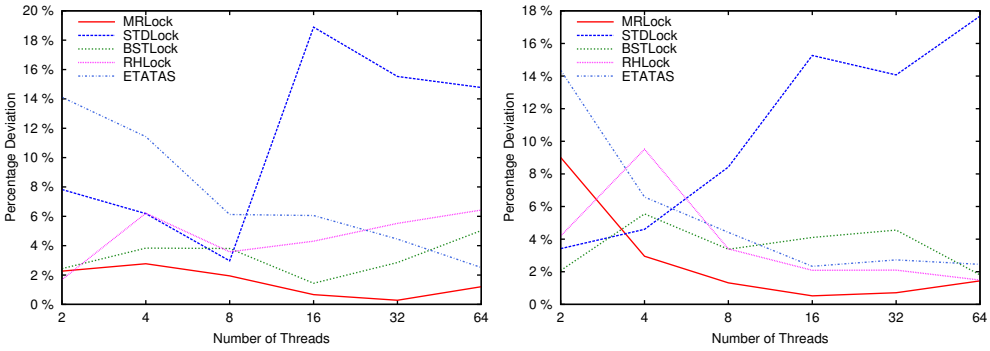
When the level of contention is low, MRLock and ETATAS do not exhibit performance advantages over the other approaches. This is shown in Figure 2d. In this scenario we observe that when using 32 threads, MRLock is 3.7 times slower than STDLock. The difference in performance decreases to about 2 times on 64 threads, which implies that our approach has a smaller scaling factor. We also observe better scalability of the MRLock approach against ETATAS; when moving from 32 threads to 64 threads the performance of ETATAS degrades threefold resulting in a 2 times slowdown compared to MRLock.

The contention level that is a pivot point for our algorithm's performance is about 12.5% as shown in Figure 2e. MRLock ties with RHLock and outperforms all other algorithms. MRLock is 4 times faster than STDLock and twice as fast as ETATAS on 64 threads. In addition, MRLock exhibits better scalability compared to its alternatives. The time needed to complete the benchmark for ETATAS, BSTLock and STDLock almost tripled when the number of threads is increased from 32 to 64, while the time of MRLock only increases by 100%. In Figure 2f, STDLock takes more than 20 times longer than MRLock. MRLock outperforms all other methods on all scales except for the ETATAS.

Overall, MRLock exhibits high scalability on all levels of contention, it outperforms STDLock and BSTLock 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 ETATAS when the number of thread are small, it outperforms the ETATAS by at least 2 times on 64 threads.

5.4 Performance Consistency

It is often desirable that an algorithm produces predicable execution time. We demonstrated in Section 5.2 that our multi-resource lock exhibits reliable execution time regardless the level of resource contention. Here, we further illustrate that our lock implementation achieves more consistent timings among different runs when compared to the competing implementations. Figures 3b and 3a display the percentage deviation of execution times from 10 different runs. Since we generate randomized resource requests at the beginning of each test run, the resource conflicts is different for each run. We show the deviation normalized by the average execution time of each approach on the y-axis, and the number of threads on the x-axis. Overall, MRLock produces the smallest deviation that is often within 2% or its executing time. Notably ETATAS, which adopts the same batch request handling approach as MRLock, reached a maximum deviation of 18%. This indicates that the incorporation of a FIFO queue alleviated the contention and stabilized our lock algorithm.



(a) Resource contention 32/64

(b) Resource contention 32/64

Fig. 3. Normalized deviation out of 10 runs

6 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 provides FIFO fairness for contending threads, and is scalable with minimal overhead increase over the best available solutions. As demonstrated by our performance evaluation, the MRLock algorithm exhibits reliability and scalability that can be beneficial to applications with high contention or when system scalability is desired.

Possible extension for this algorithm includes creating a NUMA aware algorithm by adopting the hierarchical queue structure and an adaptive method to choose from several locking algorithms based on the level of contention in a system.

Acknowledgment. This material is based upon work supported by the National Science Foundation under CCF Award No.1218100. The authors would like to thank Dimitry Vyukov for providing insightful implementation tips on the non-blocking queue.

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. IEEE Transactions on Parallel and Distributed Systems 1(1), 6–16 (1990)
3. Awerbuch, B., Saks, M.: A dining philosophers algorithm with polynomial response time. In: Proceedings of the 31st Annual Symposium on Foundations of Computer Science, pp. 65–74. IEEE (1990)
4. Bar-Ilan, J., Peleg, D.: Distributed resource allocation algorithms. In: Segall, A., Zaks, S. (eds.) WDAG 1992. LNCS, vol. 647, pp. 277–291. Springer, Heidelberg (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. Craig, T.: Building fifo and priorityqueuing spin locks from atomic swap. Tech. rep., Citeseer (1994)
7. Dechev, D., Pirkelbauer, P., Stroustrup, B.: Lock-free dynamically resizable arrays. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 142–156. Springer, Heidelberg (2006)
8. 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)
9. Dijkstra, E.: Hierarchical ordering of sequential processes. Acta Informatica 1(2), 115–138 (1971)
10. 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)
11. 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)
12. Fischer, M., Lynch, N., Burns, J., Borodin, A.: Resource allocation with immunity to limited process failure. In: 20th Annual Symposium on Foundations of Computer Science, pp. 234–254. IEEE (1979)
13. Harris, T.L., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 265–279. Springer, Heidelberg (2002)
14. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21(2), 289–300 (1993)
15. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann (2012)
16. 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)
17. 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)
18. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems (TOCS) 9(1), 21–65 (1991)
19. 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)
20. Raynal, M.: A distributed solution to the k-out of-m resources allocation problem. In: Dehne, F., Fiala, F., Koczkodaj, W.W. (eds.) ICCI 1991. LNCS, vol. 497, pp. 599–609. Springer, Heidelberg (1991)
21. Raynal, M., Beeson, D.: Algorithms for mutual exclusion. MIT Press (1986)
22. 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, PPoPP 2001, pp. 44–52. ACM (2001)