Design of a minimal-contention lock: M-lock
Technical Report CS-PM-2018-08-19

Philip Machanick
Department of Computer Science
Rhodes University
p.machanick@ru.ac.za

19 August 2018

Abstract

As multicore CPUs become more common, scalable synchronization primitives have wider use and ideas previously used in large-scale computation are worth re-opening for wider use. In this paper I explore one approach to scalable sychronization, a minimal-contention lock (M-lock). The key idea is to avoid spinning on a global variable but instead for each blocked task (process or thread) to spin on a local lock representing the task that immediately preceded it in attempting to acquire the lock. This creates an ordering based on the order in which tasks attempt to acquire the lock, preventing starvation. The only globally shared variable is a pointer to the next local lock to be contended for. Each contending task swaps the value of this pointer for a pointer to its own variable. It spins on the variable previously pointed to by the global pointer. Each waiting task spins on a lock only seen by itself and the owner of that lock variable. While a task is spinning, the lock variable can be held in its local cache until invalidated by the lock owner when it releases the lock. Consequently, the amount of bus traffic is considerably less than with a spinlock, which has the pernicious feature that the task releasing the lock is delayed by all the other bus traffic arising from contention for the lock. An MCS lock has similar properties but is more complicated and requires more memory contention-causing operations. This report outlines the design of the M-lock and points to likely performance gains.

1. Introduction

This report outlines the design of a minimal-contention lock, M-lock. The design draws on the ideas of the widely used and cited MCS lock [5, 2, 12], named after the initials of its creators [9] and further reduces the amount of contention.

Intel's Xeon E7 line has up to 24 cores and their Xeon Phi Knights Landing has 72 cores [13], foreshadowing that level of available parallelism in the consumer space. While considerable work was done in the 1990s on reducing memory contention, that was in the context of relatively high-end machines. As multicore and many-core CPUs become more common, there is a growing need for scalable synchronization for a wider user base – not just high-performance computing (HPC).

In this report, I start with an outline of the MCS lock, then I describe an approach of going back to first principles to design a minimal-contention lock (M-lock – which can stand for "minimal-contention lock" or my surname initial if it proves not to be minimal). My strategy is to start with the memory accesses needed to implement a lock. This is more easily seen in a simple RISC machine code version than in a high-level language or pseudocode. I then derive a C version of the algorithm. I wrap up with a discussion of potential performance gains.

2. MCS lock

The problem that the MCS lock solves is the lack of scalability of any lock that requires contending for a shared variable – such as a spinlock, which can be summarized as in Figure 2.1. A spinlock is inherently not scalable as it puts each task into a tight loop that causes invalidations of the lock variable out of each other contending task's[1] local cache, followed by a write miss. Releasing a spinlock has pernicious effects too, including delaying the task that holds the lock in its own attempts at accessing memory, if it updates a shared variable not already in its own cache. Releasing the lock adds to contention for memory. A spinlock also is not fair – winning the race to lock is unrelated to the order tasks first attempted to acquire the lock. MCS also partially solves the fairness problem.

The MCS lock algorithm is presented in Figure 2.2. The algorithm implements a linked list of spinning tasks.

Acquiring a lock requires atomically swapping a pointer at a global location with a pointer to a local instance of a lock variable. If the global pointer was NULL, that means the queue was empty so the lock is acquired. If the queue is not empty, the task inserts itself into the queue and spins on its own instance of the lock variable, which must be released by its predecessor when it leaves the critical section.

In order to implement FIFO ordering, a compare-and-swap atomic operation is required [9].

There are other ways to achieve FIFO ordering, a requirement to ensure fairness (or to prevent starvation). For example, a ticket lock [9, 3] provides each waiter with a number and it spins on a central lock variable until its number comes up; this creates a FIFO ordering but does not remove the lock hotspot as it is still spinning on a global shared variable. However, the difference between a waiting thread's ticket and that of the lock-holder's ticket can be used to implement proportional wait, reducing memory traffic from spinning.

A detailed study of ticket locks and MCS locks confirms that the latter design is more scalable [6]; another study showed that the ticket locks used in the Linux kernel cause performance collapse with as few as 48 cores [1]

MCS has a number of useful features – it only requires two atomic memory operations, one to acquire and one to release. It also only spins on a local value while waiting to acquire the lock.

On the other hand, it has some weak points. On release, a compare-and-swap is required if the releasing task is not the last in the list. If this operation is not available, the coding is a little more complex and does not guarantee FIFO ordering. It also needs to spin if the next element of the list is not the last item but has not yet set its next pointer. The code is difficult to understand and there are subtle edge conditions that make proving correctness difficult [7].

So: can we do better?

```
typdef uint8_t LockT;

void simpleacquire (LockT *lock) {
  volatile bool mylock = 1;
  do {
    mylock = swap (lock, mylock);
  } while (mylock);
}

void simplerelease (LockT *lock) {
  *lock = 0;
}
```

Figure 2.1: Spinlock. *It uses a* `swap` *atomic memory operation that takes a pointer to a lock value and a lock value. It replaces the value pointed at by the first parameter by the value of the second parameter and returns the value that was overwritten. In all code examples I leave out details like padding to avoid false sharing and processor-specific coding to reduce penalties of spinning.*

[1]I use the Linux terminology where a task can either be a process – in this case, cooperating with others using shared memory – or a thread.

```
typdef struct {
  LockT * next;
  bool locked;
} LockT;

void mcs_acquire (LockT *lock, LockT * I) {
  I->next = NULL;
  LockT * pred = swap_ptr (lock, I);
  if (pred) { // queue not empty
    I->locked = true;
    pred->next=I;
    while (I->locked) ; // spin
  }
}

void mcs_release (LockT *lock, LockT * I) {
  if (!I->next)
    if (compare_and_swap (L, I, NULL))
      return;
    while (!I->next); // spin
  I->next->locked = false;
}
```

Figure 2.2: MCS algorithm [4]. *It requires a* `compare_and_swap` *atomic memory operation to ensure FIFO ordering. Without this primitive, the release algorithm is more complicated and does not guarantee FIFO ordering. Atomic operations:* `compare_and_swap` *swap and returns true if the stored value changed;* `swap_ptr` *atomically swaps two pointer values: it returns the old value of the pointer that changed in memory.*

An improved lock algorithm should have fewer atomic memory operations than MCS, avoid spinning on release and be ensure FIFO ordering without requiring any atomic operations that are not widely available. It should also not result in any more shared-memory transactions, particularly invalidations.

3. M-lock

The fact that MCS has been something of a gold standard for over 25 years suggests that it is inherently a good idea. However, that does not mean it is impossible to do better. The approach I take is to consider the memory operations needed for a lock, minimizing global updates. From this I derive a machine-code approach and from there, an algorithm expressed in C.

I outline design aims then set up the basis for a solution. From this start, I develop a basic idea, then correct for an *ABA problem.* An ABA problem arises when a shared variable is set to A then B then A again and a task only sees the two instances of A, incorrectly interpreting this as no change in the variable [10]. I finally present a C version of the lock to demonstrate that it can be implemented in a high-level language and to make it easier to compare with the MCS algorithm.

3.1 Aims

My aims, counting operations over both acquire and release, are:

- *commonly available atomic operations* – ideally, only an atomic swap as that is widely available or can be implemented out of other primitives
- *minimal atomic memory operations* – ideally, no more than one
- *minimal global updates* – ideally, no more than one
- *minimal spinning* – ideally, only one one variable only seen by two tasks, the waiter and the lock holder
- *FIFO ordering* – to ensure fairness
- *simplicity* – to facilitate checking or proving correctness

By this standard, MCS falls short. It can only achieve FIFO ordering if a compare-and-swap is available, it needs up to two atomic primitives (one for acquire, sometimes one for release) and it can also spin on release.

3.2 Starting point

A global pointer swap is a good starting point but there is no need to implement a full linked list. Each task only needs to know where to find its own local lock variable and that of its predecessor on which it spins.

In order to develop the design, I use a simplified RISC instruction set:

- *zero register* – register `R0` is hardwired to zero
- *load word* – `lw R1, offset(R2)` – copies the value at the address `offset`+*contents of register* `R2` to register `R1`
- *store word* – `sw R1, offset(R2)` – copies the value in register `R1` to the address `offset`+*contents of register* `R2`
- *add* – `add R1, R2, R3` – add values in registers `R1` and `R2`, putting the answer in `R3`
- *add immediate* – `addi R1, R2, val` – add value in register `R1` and the 16-bit signed value `val` embedded in the instruction, putting the answer in `R1`
- *atomic swap* – `swap R2, 0(R1)` – copies the value in register `R2` to address `offset`+*contents of register* `R1`, replacing the contents of `R2` by the old contents of the memory location
- *branch on zero* – `bez R1, label` – if `R1` contains zero, branch to the label
- *branch on not zero* – `bnez R1, label` – if `R1` does not contain zero, branch to the label

3.3 Basic M-lock

An M-lock has a single lock variable in each contending task as well as a spare one that is used to set up the lock so the initial attempt at swapping the global pointer will point to this spare variable that is initialized to 0 (unlocked).

Setting up the M-lock requires initializing the spare lock variable to 0 and setting the global pointer to point to it. Then each local lock variable that will be spun on by one other task next in line must be initialized. Since the local lock pointer will be swapped with the global one, it is also necessary to keep a copy of it. Register use is as follows:

- *temporary for constants* – register `R1`
- *address of local lock value* – `R2`
- *saved address of local lock value* – `R3`
- *address of global lock pointer* – `R4`
- *value of lock we spin on* – `R5`; this is loaded from the address we pick up from `R4`

The spare lock variable is initialized to zero and the global pointer set to point to it before any task can attempt to access the lock; this is best done before tasks (aside from the parent) are launched; since this is separate code run once at initialization, register `R2` can be recycled for this purpose:

```
        sw R0,0(R2)  # local lock value to memory
        sw R2,0(R4)  # spare lock address set up
```

Code to set up a lock variable in a task is as follows:

```
        addi R1,R0,1 # to initialize local lock
        sw R1,0(R2)  # local lock value to memory
        addi R3,R2,0 # save our lock address
```

We can now attempt to acquire the lock:

```
        swap R2,R0(R4),1 # swap lock addr with global
test:   lw R5,0(R2)
        bnez R5,test # not zero? spin
        addi R2,R3,0 # restore our lock address
```

At the end of this code, we have the lock and have the lock address back in its original register for next time and for releasing the lock, which is done as follows:

```
        sw R0,0(R2),R0
```

Compared with a spinlock, there is a little more setup code and we need $N + 1$ lock variables for N tasks, in addition to one global lock variable (now containing an address instead of a zero or one value). The spinning code is the same number of instructions – a load and a branch as opposed to an atomic swap and a branch[1]. Releasing the lock is a store as before.

The really big difference comes in shared memory contention and cache misses. For the spinlock, every time through the loop, if more than one task is spinning, each one will invalidate any other cache's copy of the shared lock variable after experiencing a cache miss and attempting to write the lock value (a consequence of the atomic swap). For the M-lock, there is contention for swapping the lock variable pointer but once past that point, each task will only have one cache miss for the lock variable it needs to spin on and *it does not invalidate it out of the owner's cache because it only reads it.*

3.4 Refined M-lock

The implementation as given so far would work if the lock was only ever held once. However, if the critical section needs to be entered a second time, an ABA problem could arise. For example, if two tasks, T_a and T_b (respectively with lock variables L_a and L_b), attempt to enter the critical section and the following sequence of events occurs:

- T_a acquires the lock
- T_b spins on L_a

[1]Noting that the swap could be replaced by two instructions in some instruction sets.

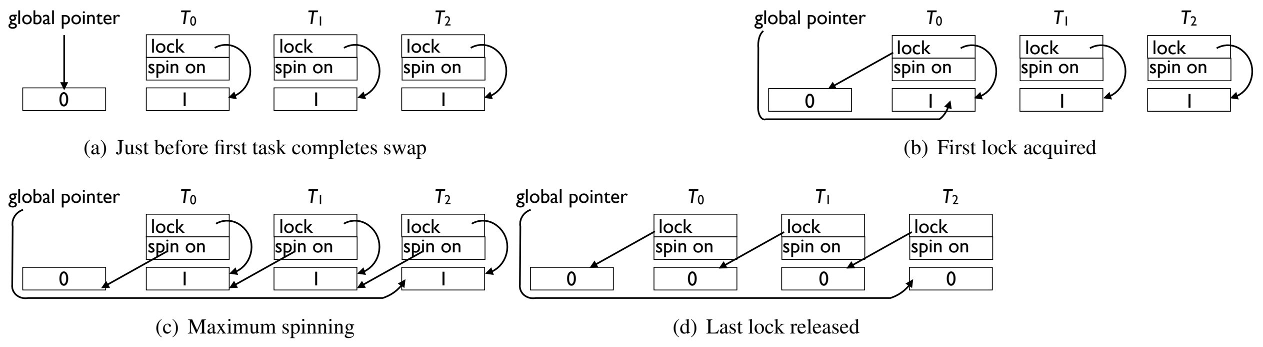


Figure 3.1: State of lock variables at the start and finish of a critical section. *Tasks are numbered in the oder that they attempt to acquire the lock. Each task swaps the global pointer with the pointer to the task's local lock. For T_0, this atomic swap is shown in two stages. When T_2 releases the lock, provided there is no other attempt at acquiring the lock, the global pointer points to what was previously the lock of T_2.*

- T_b is suspended by the operating system for any reason when it last sees the lock as set (value 1)
- lock L_a is released (set to 0)
- T_a tries to acquire the lock again while T_b remains suspended (set to 1)
- T_a swaps the global lock pointer and starts spinning on L_b
- T_b is scheduled again and sees L_a as 1 and continues spinning

It is not necessary that T_b be suspended; condition where T_a is able to set the lock to 0 then back to 1 before T_b sees it as 0 is sufficient.

A simple fix to this is to hand the lock location over to the task spinning on it. So the local lock starts as belonging to the task that sets it and ends up belonging to the task that was spinning on it after it is released. The initialization code is unchanged; we still need to remember our old lock variable's address to release it.

Attempting to acquire the lock is the same except we do not restore the old lock address at the end:

```
         swap R2,R0(R4),1 # swap our lock addr with global
test:  lw R5,0(R2)
       bnez R5,test # not zero? spin
```

We release the lock now using the saved address of our old lock:

```
sw R0,0(R3),R0
```

If we now want to acquire the lock again, we do so with the address in `R2`, which is no longer owned by the task that beat this one to the lock. In the two-task scenario, T_a would have the address initialized as the spare lock variable in `R2`, set the contents to 1 and swap that for the global lock address, which would be the lock still held by T_b. This eliminates the ABA problem.

Figure 3.1 illustrates the state of the lock at various stages:

- Figure 3.1(a) shows the state just before T_0 swaps its lock pointer address with the global pointer
- Figure 3.1(b) is when the swap has been completed for T_0
- Figure 3.1 illustrates the maximum amount of spinning
- Figure 3.1(d) shows the state after all tasks release the lock

If a task completes, it is free to deallocate its instance of the lock variable – the previous owner loses its interest in it as soon as it sets it to 0, releasing the lock. The global pointer always points at a lock variable for the task that most recently tried to acquire the lock and is set to zero when no one holds the lock; at that stage, no task points to this variable so it can also be deallocated if the lock is no longer in use.

```
typedef struct {
    volatile uint8_t lockval;
} m_lock_t;

typedef struct {
    m_lock_t
      ** global, // points to global lock
      * mylock,  // points to lock owned by me
      * spinon;  // points to lock I spin on
} m_lock_node_t;

void m_acquire (m_lock_node_t * lock) {
    lock->spinon = swap_ptr(lock->global,lock->mylock);
    while (lock->spinon->lockval != 0) ; // spin
}

void m_release (m_lock_node_t * lock) {
  lock->mylock->lockval = 0;
  lock->mylock = lock->spinon;
  lock->mylock->lockval = 1; // ready for next time
}
```

Figure 3.2: M-lock in C. *Initial lock values: 1 local, 0 global. Each local lock structure contains a pointer to the global pointer;* `swap_ptr` *is as in Figure 2.2.*

3.5 M-lock in C *vs*. MCS

Figure 3.2 illustrates a C implementation. Compare it with Figure 2.2 to see how much simpler M-lock is. The only additional cost is more pointers for each local data structure. However, in practical terms, any lock structure needs to be padded to fit the size of a cache line to avoid *false sharing* [14], which arises because caches are managed at block (cache line) granularity. So M-lock will not require more storage than MCS in a typical scenario: for 64b pointers, 4 can fit into a 32B cache line, or 8 in a 64B cache line.

This implementation meets all my aims. It only uses atomic swap. It only has one atomic memory operation (in acquire, none in release). It only has one global update, the swap of the global pointer. Spinning is on a variable seen by only two tasks – the waiter and the lock holder. It ensures FIFO ordering and the implementation is much easier to understand than MCS without subtle edge conditions.

I expect that M-lock will scale better than MCS especially under high contention when spinning on release of MCS is more likely. Since M-lock is simpler to code, it is easier to be sure it is right, whether using informal or formal methods.

4. Performance

I do not present detailed performance analysis here; that is the work of future research. Using a simple model for a specific scenario, I show that an M-lock under high contention (all tasks arrive at the lock simultaneously), M-lock results in the minimum number of shared-memory transactions, assuming a multicore system in which sharing is through the lowest-level cache (LLC) [8].

The outer-level repeats are designed to hide the latency of Pthreads thread invocation and the barrier. The inner loop is simulates a nontrivial critical section; it should have one miss the first time it updates the shared variable (the array index could be varied for other memory use patterns).

Ideally acquiring and releasing the lock should at worst scale linearly in the number of threads, N, since the lock serializes the shared update. Since my shared updates are $O(1)$ per thread, the overall transaction should at worst scale as $O(1)$.

4.1 Scalability Test

A *scalable lock* is one that adds at most constant time per additional thread. In this case, that means shared update of a constant number of variables scales no worse than $O(N)$. In addition, scalable locks can be ranked by the magnitude of the overhead they add. Scalability cannot be measured in isolation from a workload as the shared update in the critical section also writes to memory and contention for the lock is in competition for those shared updates.

I use here a simple test of scalability: the following code (with variations in the lock interface) to measure latency of several types of lock is launched as N threads using Pthreads:

```
void *WORK (void *threadDataPtr) {
    pthread_barrier_wait(&barrier);
    for (long int j = 0; j < 100000; j++) {
        ACQUIRE
        for (long int i = 0; i < 10; i++) {
            shared[0]++;
        }
        RELEASE
    }
}
```

The main program does not participate in the workload, but waits for the N threads it launches using a Pthread join.

I use the implementation of MCS locks, ticket locks and hierarchical ticket locks of David *et al.* [4] and my own implementations of a spinlock and my own synchronization primitives, M-locks, M-barriers and barlocks.

Coding is in C, using the gcc tool chain with the following compiler options:

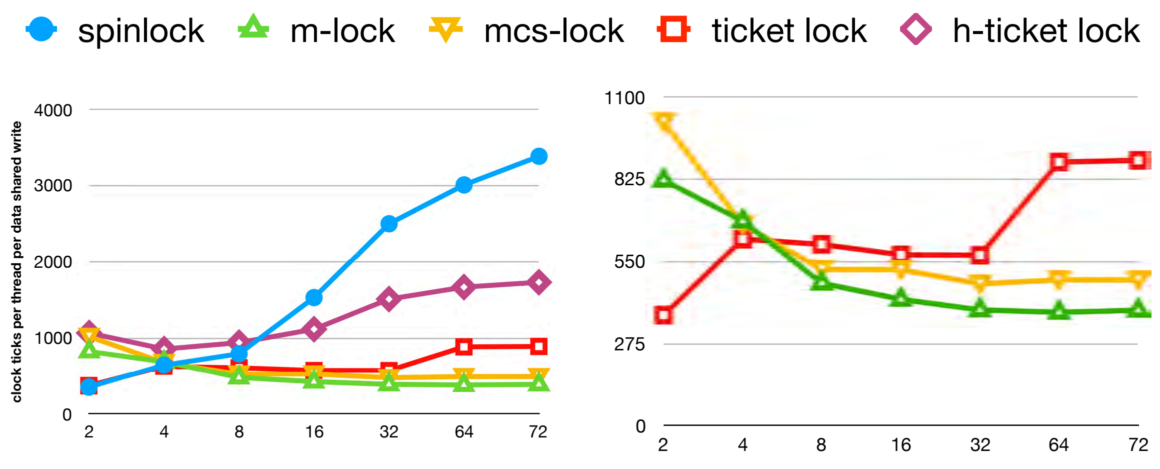
```
-DXEON -flto -D_GNU_SOURCE -lrt -lpthread -lnuma
```

The `-DXEON` option triggers CPU-specific options in the source provided by David *et al.*, which I alter for the change in hardware configuration of my Xeon system. I use gcc 5.4.0-6 on Ubuntu 16.04.9.

4.2 Hardware constraints

The machine available to me is configured as follows:

- 2 sockets, each with a 2.1GHz Intel Xeon E5-2695 v4
- each socket: 18 cores, 2 threads per core
- caches:



(a) *all locks* (b) *most scalable locks*

Figure 4.1: Preliminary scalability test. *Timing is scaled to the number of shared memory operations, the thing of interest to the programmer.*

  - L1d, L1i: 32KiB each
  - L2: 256KiB
  - L3: 45MiB (shared)

- 128GB DRAM

Because of the available hardware resources, I run variants of up to 72 threads. With a real workload, this may be too much as the machine has 36 actual cores and hardware threads are limited by the extent to which multiplexing the pipeline makes sense.

4.3 Measurement

Figure 4.1(a) illustrates a run of this workload; timing is scaled to the number of shared accesses and the number of threads, i.e, $\frac{t_{total}}{R \times I \times N}$ where t_{total} is the total run time from thread launch until all have joined, $R = 100000$ is (outer loop) repetitions, $I = 10$ is (inner loop) iterations and N is the number of contending threads. The logic for this calculation is that it captures the cost of lock and shared memory contention.

Spinlocks and hierarchical ticket locks scale worst, the latter possibly because this configuration does not have a high inter-socket penalty. To make clearer how the more scalable locks perform, Figure 4.1(b) leaves out hierarchical ticket locks and spinlocks. With this workload, M-locks do best from 8 threads upwards. With up to 4 threads, the less scalable locks come out ahead because they have less overhead. For 4 or 8 threads, MCS and M-locks differ by a small amount; or all other cases, M-locks are at least 20% faster.

4.4 Summary

The scalability test is a very limited performance test – it indicates the possibility that M-locks scale significantly better than MCS locks but more anaylsis is needed, as is a wider range of workloads that capture the variability in both user-level and kernel code. For the specific example measured, MCS and M-locks clearly scale better than the others, particularly for $N > 32$.

5. Conclusions

M-lock is simpler and inherently more scalable than MCS; performance analysis, while very preliminary, supports this – pointing to a need for more credible scalability studies.

How did I get there? I started from first principles, considering only memory accesses, which I represented in a simplified RISC instruction set. Had I started from C code or some other high-level language, it is doubtful that I would have found a better alternative than MCS.

The lesson? If you are trying to find an approach that is a clean fit to a computer architecture concern, a high-level language may not be the right abstraction: starting at machine code level may lead you to a better answer.

Where from here? It would be useful to compare performance in user-level code with mutexes, which do not spin but us a wait and signal approach to avoid wasting CPU time, and avoid starvation by queueing waiters [11]. Spinning, except for very short durations, carries the risk of degrading a task's priority if the scheduler mistakes spinning for useful work. While mutexes have significantly more overhead, if wait times significantly exceed system call overhead, they could come out ahead. Implementing in a kernel is a little more difficult because the M-locks API does not match that of e.g. Linux kernel locks.

Finally, proving M-locks correct would be a useful exercise.

References

[1] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[2] Austin T Clements, M Frans Kaashoek, Nickolai Zeldovich, Robert T Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems*, 32(4):10, 2015.

[3] Travis Craig. Building FIFO and priority queuing spin locks from atomic swap. Technical report, University of Washington, Seattle, 1993. URL: `ftp://trout.cs.washington.edu/tr/1993/02/UW-CSE-93-02-02.pdf`.

[4] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 33–48. ACM, 2013.

[5] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys*, 43(4):35, 2011.

[6] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *OSDI*, volume 16, pages 653–669, 2016.

[7] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. Safety and liveness of MCS lock – layer by layer. In *Asian Symposium on Programming Languages and Systems*, pages 273–297. Springer, 2017.

[8] Philip Machanick. A preliminary study of minimal-contention locks. In *Proceedings of SAICSIT 2018*, September 2018. In press.

[9] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991. URL: `http://doi.acm.org/10.1145/103727.103729`.

[10] Maged M. Michael. The balancing act of choosing nonblocking features. *Commun. ACM*, 56(9):46–53, September 2013. URL: `http://doi.acm.org/10.1145/2500468.2500476`, `doi:10.1145/2500468.2500476`.

[11] Bradford Nichols, Dick Buttlar, Jacqueline Farrell, and Jackie Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly, Sebastopol, CA, 1996.

[12] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *41st International Symposium on Computer Architecture*, ISCA, pages 265–276. IEEE, 2014.

[13] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation Intel Xeon Phi product. *IEEE Micro*, 36(2):34–46, 2016.

[14] Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

Acknowledgements

Financial support: NRF grant 98952.