Mutual Exclusion - Two linear wait software solutions

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Abstract

More than 50 years after Dijkstra first formulated the mutual exclusion problem, the research on software solutions to solve this problem is still ongoing. Over the years, researchers like Dekker, Dijkstra, Knuth, Peterson, Lamport, and many others, have proposed solutions with different properties, ranging from algorithms with high unfairness, to starvation-free algorithms with linear wait.

Most of the recent CPU architectures already provide special instructions which allow to easily implement mutual exclusivity with algorithms such as MCS, CLH, Ticket Lock, or Tidex Lock, but there are still many CPUs without such instructions, for which a software solution is required.

Our research focused on developing algorithms that used a minimum number of writes to shared memory, with one write before entering the critical section, and another write after leaving it. This led to two new starvation-free algorithms which we named CR-Turn and CRHandover. We present experimental results for microbenchmarks with high and low contention, and in four different architectures: AMD x86-Opteron, Intel x86-Haswell, PowerPC, and ARMv7. The results show that for both low and high contention scenarios, these two algorithms are amongst the best of the starvation-free in the four different architectures.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Mutual Exclusion

General Terms Algorithms, Design, Performance

Keywords mutual exclusion, software solution, locks

1. Introduction

Mutual exclusion is one of the first concurrency problems to be tackled [7], and today, it is still a subject of research [2]. In multiprocessor systems with shared memory, the techniques using the special test-and-set instruction or compare-and-exchange instruction — like CLH [5, 15], MCS [17], Ticket Lock [16] or Tidex Lock [22] — can be effective, but there are to this day CPUs that do not provide such instructions, and for those, we can only rely on algorithms that use simple store and load instructions.

The mutual exclusion problem is described as N threads/processes each executing a critical and a noncritical section, the restriction being, that from the moment a process enters its critical section and until it leaves it, no other process is allowed to execute their own critical section. A critical section of a process is a section of its program, for which we wish to ensure is never executed at the same time that any other process is executing its own critical section [20]. A solution to the critical section problem needs to satisfy two important properties:

1. *Mutual exclusion*: There will be at most one process executing the critical section at a time, or as formally stated by Lamport [14], for any pair of distinct processes i and j, no pair of operation executions $CS_i^{[k]}$ and $CS_j^{[k]}$ are concurrent.
2. *Deadlock Freedom*: The critical section will not become inaccessible to all processes. This means that if a number of processes attempts to execute their critical sections, then after a finite amount of time some process will be allowed to do so.

These two properties combined, guarantee that in a finite number of steps, *one and only one process* will have access to a critical section at a time. Later, the property of *starvation-freedom* was introduced that guarantees *all processes* will access the critical section in a finite number of steps. Other properties are stated by Dijkstra [8]:

- The solution must be symmetrical between the N computers; as a result, we are not allowed to introduce a static priority.
- Nothing may be assumed about the relative speeds of the N computers; we may not even assume their speeds to be constant in time.
- If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others.

Dijkstra [7] was the first to formulate the mutual exclusion problem between only two processes, and it is attributed to Dekker [11] the first known solution to this problem. Later, Peterson [19] provided a simpler and more efficient solution for two processes, using 3 variables where there is a write race on the variable `turn`, followed by Kessels's [10] solution using 4 variables instead of 3 where each process modifies its own variables and the turn is decided based on the combined result of both variables.

The first solution considering N concurrent processes was provided by Dijkstra, and was a source of inspiration to several solutions that came after. Knuth [11] was the first to realize that there was no guarantee that a process could not wait indefinitely to access the critical section, introducing the notion of fairness. His solution is starvation-free but in the worst case scenario a process could be overtaken $(2^{(N-1)} - 1)$ turns. Later, DeBruin [6] improved Knuth's solution by guaranteeing a process would only have to wait $N(N-1)/2$ turns. In this same approach, Eisenberg and McGuire [9] finally provided a solution that guaranteed a maximum wait of $N - 1$ turns, where each preceding winning process gives the turn to the next process that is attempting to enter its critical section.

Another class of algorithms which satisfies the first-come first-served (FCFS) property was introduced by Lamport [13] and he further considered a special property, RW-safe, where reads and writes to the same memory location could execute simultaneously and reads could return any arbitrary value and still the algorithm would execute correctly. Hehner [21] has improved the performance of Lamport's algorithm also using less memory, but it no longer satisfied the RW-safe property.

A different approach was to develop algorithms that are tournament based, where each tournament opposes D processes (with D greater than one). The tournament is established following a logical tree where each node is a round that opposes D processes. The algorithm chosen for each round has to guarantee mutual exclusion between those D processes, and typically when D is two, the algorithm used can be Dekker's, Peterson's or Kessels's. The tournament based approach has an advantage in cases where a process is delayed for some time, all other processes that are competing in a different branch of the tree will not get delayed by this process and, subsequently, these processes can overtake the delayed process an unbounded number of times. This is beneficial in scenarios with over-subscription, where the number of processes is greater than the number of cores.
Peterson and Fischer [19] devised a tournament solution where the logical tree is determined using the bits of each process identification number. On the other hand, Taubenfeld [1] performs a direct tree walk where on each node a Peterson 2-Process algorithm is used to determine the process that will move on to the next level on the tournament. Both Peterson and Taubenfeld's solutions use a maximal binary tree. Kessels also performs a direct tree walk but instead uses a minimal binary tree with Kessels 2-Process algorithm. Recently, Buhr [2] has used Kessels direct minimal tree walk with Peterson 2-Process algorithm for contention on each node, with the additional optimization using a precomputed tree that is accessed as a table lookup.

Finally we will refer to Burns and Lynch algorithm [3] and Lamport's One Bit solution [14], since our first approach on solving the mutual exclusion problem has led us to the same algorithm Lamport has described. Lamport algorithm is not starvation free, and is designed with the following rule: processes with lower priority will give up while processes with higher priority will wait for lower priority processes to give up or leave the critical section (in case those processes have not seen that a higher priority process has arrived). It is clear that an higher priority process can continuously starve a lower priority process. Starting from this algorithm, we have developed another two solutions, which guarantee starvation free with linear wait.

In this paper we will present the following innovations:

- A starvation free solution with linear wait using a global variable turn, and a minimum number of release-stores [4] of 2, one before entering the critical section and one after leaving it.
- A starvation free solution with linear wait using a distributed array for the handover, and a minimum number of release-stores of 2.

Table 1 shows the starvation freedom properties of different algorithms, and the number of states per thread.

We present our CRTurn and CRHandover starvation-free algorithms in section 2, followed by experimental results with a microbenchmark on four different CPU architectures in section 3, and conclude in section 4.

2. Algorithms

The algorithms described in this section are presented in C11 code. The goal of using C11 is to provide source code that is cross-platform and known to be correct regardless of the target architecture or operating system. We chose C11 over C++11 because these algorithms are more likely to be used on small embedded devices, where C is still the language of choice by most engineers, and also, installing a working C++1x compiler in some of these systems proved difficult as some of the tools and distributions are not mature enough.

	Free from Starvation?	States per Thread
Dekker	linear wait	2
Dijkstra	no	3
Knuth	exponential wait	3
Peterson	quadratic wait	2N
Hehner	linear wait	N
Lamport - One Bit Solution	no	2
Burns & Lynch	no	2
Eisenberg & McGuire	linear wait N	3
Szymanski's	linear wait, N	5
Peterson - Buhr	unbounded	2
Correia & Ramalhete Turn	linear wait, N	3
Correia & Ramalhete Handover	linear wait, N	3

Table 1. Comparison table between the different mutual exclusion algorithms, where N is the number of active threads

2.1 CRTurn

The CRTurn algorithm starts from an highly unfair algorithm which we arrived at independently but it is actually Lamport's *One Bit Solution* [14], refered to in Buhr's paper [2] as LamportRetract. It consists of an array `states[]` where each entry has been assigned to a specific thread, and there are two possible states: `UNLOCKED` and `LOCKED`, where `UNLOCKED` is the default value.

The LamportRetract algorithm can be modified to provide starvation-free guarantees. To achieve that, we must introduce the following changes: first, there needs to be three states instead of two, which we named `UNLOCKED`, `WAITING` and `LOCKED`; second, we need a global variable which we named `turn`. The C11 source code for this algorithm can be seen in Algorithm 1, and the methods `validate_left()` and `validate_right()` are shown in Algorithm 2.

Algorithm 1 CRTurn Algorithm

```
1  atomic_store(&states[id], LOCKED);
2  while (1) {
3      int lturn = atomic_load(&turn);
4      if (!validate_left(id, lturn)) {
5          atomic_store(&states[id], WAITING);
6          while (1) {
7              if (validate_left(id, lturn) && lturn == atomic_load(&turn)) break;
8              Pause();
9              lturn = atomic_load(&turn);
10         }
11         atomic_store(&states[id], LOCKED);
12         continue;
13     }
14     while (lturn == atomic_load(&turn)) {
15         if (validate_right(id, lturn)) break;
16         Pause();
17     }
18     if (lturn == atomic_load(&turn)) break;
19 }
20 CriticalSection( id );
21 int lturn = (atomic_load_explicit(&turn, memory_order_relaxed)+1) % N;
22 atomic_store_explicit(&turn, lturn, memory_order_relaxed);
23 atomic_store(&states[id], UNLOCKED);
```

In CRTurn, a thread attempting to enter its critical section, starts by changing its state from `UNLOCKED` to `LOCKED` in line 1. Intuitively, to make the algorithm starvation free, it is essential that once a thread signals its arrival with `LOCKED`, it can no longer retract to the intial state of `UNLOCKED`, thus justifying the creation of a new state `WAITING`. It then checks for other threads *to its left* that are

Algorithm 2 Helper methods for CRTurn Algorithm

```
1  int validate_left(int id, int lturn) {
2      int i;
3      if (lturn > id) {
4          for (i = lturn; i < N; i++)
5              if (atomic_load(&states[i]) != UNLOCKED) return 0;
6          for (i = 0; i < id; i++)
7              if (atomic_load(&states[i]) != UNLOCKED) return 0;
8      } else {
9          for (i = lturn; i < id; i++)
10             if (atomic_load(&states[i]) != UNLOCKED) return 0;
11     }
12     return 1;
13 }
14
15 int validate_right(int id, int lturn) {
16     int i;
17     if (lturn <= id) {
18         for (i = id + 1; i < N; i++)
19             if (atomic_load(&states[i]) == LOCKED) return 0;
20         for (i = 0; i < lturn; i++)
21             if (atomic_load(&states[i]) == LOCKED) return 0;
22     } else {
23         for (i = id + 1; i < lturn; i++) {
24             if (atomic_load(&states[i]) == LOCKED) return 0;
25     }
26     return 1;
27 }
```

in `LOCKED` or `WAITING` state by calling `validate_left()` on line 4. If all threads to the left are in `UNLOCKED` state then it remains to check to *its right side* for threads in `LOCKED` state, otherwise it must go to `WAITING` so as to yield to the threads on the left. These state transitions are summarized in the schematic of figure 1.

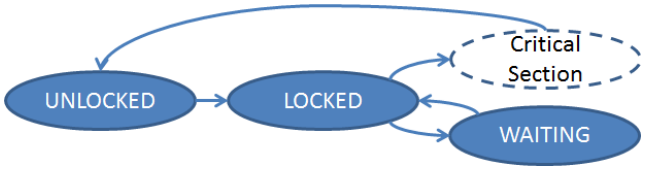

Figure 1. State machine for CRTurn and CRHandover. A thread entering its critical section must do so only from `LOCKED` state.

This is still not enough to provide starvation freedom because a thread to the left could always overtake the waiting thread, and for that, we need Eventually, after a maximum number of turns of $N-1$, the `turn` variable will equal the thread's id, which means that all other threads will yield to this one if it is in `LOCKED` or `WAITING`, thus implying that the starvation-freedom is linear with the number of threads N.

On CRTurn, the array of states functions like a ring buffer, where the `turn` variable defines the separation between the other threads that are *to the left* of the current thread and *to the right*, and a schematic is shown in figure 2. When the `turn` is below the `id` of the current thread, then the threads between `turn` and `id` are *to the left* (lines 9 and 10 of Algorithm 2), and all other threads are *to the right* (lines 18 to 21 of Algorithm 2). When the `turn` is higher than the `id` of the current thread, then the threads between `id` and `turn` are *to the right* (lines 23 and 24 of Algorithm 2), and all other threads are *to the left* (lines 4 to 7 of Algorithm2).

Unlike other algorithms with a turn variable, in CRTurn there is no write race on `turn`. The `turn` variable is modified only by the thread currently leaving the critical section, which means other threads may be reading this variable as it is being modified (read race), but no other thread will be simultaneously modifying it (write race).

Notice that in line 21 of Algorithm 1 we do a relaxed load of `turn` so that this algorithm can be easily implemented with the methods `lock()` and `unlock()` without the passing of state between these two functions, but we could replace the relaxed load with a local variable `lturn`.

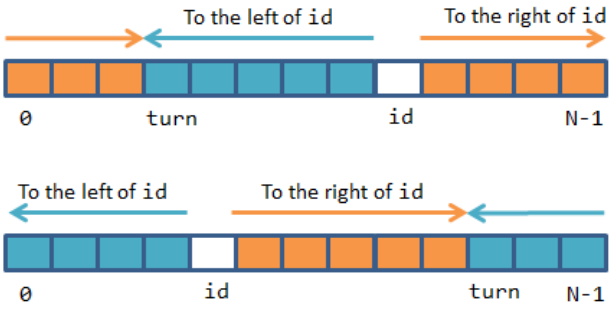

Figure 2. The array of states functions like a ring buffer, where the *left* and and *right* of the `id` are defined based on the `turn` variable.

2.2 CRHandover

Mutual exclusion lock algorithms with local spinning, like CLH and MCS, are known to provide better performance than algorithms with global spinning, like the Ticket Lock [16] or Tidex Lock [22]. With this idea in mind, we developed a variant based on the CRTurn algorithm, where instead of having a `turn` variable which changes the threads priorities, there is an `handover[]` array (with cache line padding), where each waiting thread can (local) spin waiting for its turn to enter the critical section. We named this new algorithm CRHandover and the C11 source code can be seen in Algorithm 3.

Algorithm 3 CRHandover Algorithm

```
1  int isFirstTime = 1;
2  while (1) {
3      while (atomic_load(&handoverEnabled)) {
4          if (isFirstTime) { atomic_store(&states[id], WAITING); isFirstTime = 0; }
5          if (atomic_load(&handover[id]) == MYTURN) {
6              atomic_store_explicit(&handover[id], NOTMINE, memory_order_relaxed);
7              atomic_store_explicit(&states[id], LOCKED, memory_order_relaxed);
8              goto LCS; // fast-path when under high contention
9          }
10         Pause();
11     }
12     isFirstTime = 0;
13     atomic_store(&states[id], LOCKED);
14     if (!validate_left(id)) {
15         atomic_store(&states[id], WAITING);
16         while (!atomic_load(&handoverEnabled)) {
17             if (validate_left(id)) break;
18             Pause();
19         }
20         continue;
21     }
22     while (!atomic_load(&handoverEnabled)) {
23         if (validate_right(id) && !atomic_load(&handoverEnabled)) goto LCS;
24     }
25     atomic_store(&states[id], WAITING);
26 }
27 LCS: CriticalSection( id );
28 for (int i = id + 1; i < N; i++)
29     if (atomic_load(&states[i]) == WAITING) { do_handover(id, i); goto LEND; }
30 for (int i = 0; i < id; i++)
31     if (atomic_load(&states[i]) == WAITING) { do_handover(id, i); goto LEND; }
32 if (atomic_load(&handoverEnabled)) atomic_store(&handoverEnabled, 0);
33 atomic_store(&states[id], UNLOCKED);
34 LEND:
```

On CRHandover, a thread attempting to enter the critical section will start by checking if `handoverEnabled` is true, and if it is, it will spin on its own entry in `handover[id]`, waiting for it to change by the thread currently in the critical section (line 5 of Algorithm 3), or until the handover is disabled. This code path is the *fast path* when in an high contention scenario. Otherwise, if `handoverEnabled` is false, it will act like the CRTurn algorithm where `turn` is always zero, but as soon as there are two threads contending for the critical section, the first thread that wins, will set `handoverEnabled` and `handover[otherThreadId]` to `MYTURN`, guaranteeing that the other thread is the next one to enter. Notice that at any given time that `handoverEnabled` is seen to be true,

Algorithm 4 Helper methods for CRHandover Algorithm

```
1  int validate_left(int id) {
2      for (int i = 0; i < id; i++)
3          if (atomic_load(&states[i]) != UNLOCKED) return 0;
4      return 1;
5  }
6
7  int validate_right(int id) {
8      for (int i = id + 1; i < N; i++)
9          if (atomic_load(&states[i]) == LOCKED) return 0;
10     return 1;
11 }
12
13 void do_handover(int id, int i) {
14     if (!atomic_load(&handoverEnabled)) atomic_store(&handoverEnabled, 1);
15     atomic_store(&handover[i], MYTURN);
16     atomic_store(&states[id], UNLOCKED);
17 }
```

the procedure goes into the first loop waiting for its turn to be set in the `handover[]` array.

After leaving the critical section, as soon as the next waiting thread's entry in `handover[]` is set to `MYTURN`, the turn is passed to that thread. To guarantee that the turn will not be passed a thread that already executed its critical section, we need to ensure that a thread will only enter its critical section if it set its state to `LOCKED` (line 7) beforehand, and then the turn can only be passed to a thread that is in `WAITING` state (line 14 of Algorithm 4).

Similarly to the `turn` variable in the CRTurn algorithm, the `handoverEnabled` variable in CRHandover does not have write races.

3. Experimental Results

In order to compare the performance of our algorithms with other previously known algorithms, we ran a set of microbenchmarks on multiple architectures, namely: x86 (Intel and AMD), PowerPC, and ARM. The following machines and tools were used to run the microbenchmark:

- x86: Intel Xeon E5-2666 Haswell with 2 CPUs total of 36 cores, Linux Ubuntu 14.04 64 bit, GCC 4.9.1;
- x86: AMD Opteron 6272 with 2 CPUs total of 32 cores, Windows 7, GCC 4.9.2 (Mingw);
- ppc: PowerPC Power8E with 8 cores, Ubuntu Linux 14.04 64 bit, GCC 4.9.1;
- arm: ARMv7 Marvell Armada 370/XP with 4 cores, Ubuntu Linux 15.04 Alpha, GCC 4.9.1;

Recently, with the advent of memory models on modern programming languages the one in C11/C++1x, the algorithms can be written once, and safely ran on any CPU architecture without fear of incorrect results due to re-ordering on specific architectures. Due to this, we used the benchmark developed by Buhr, Dice and Hesselink [2] and available on Github [18], but ported it to the C11 language. In their paper, the tournament techniques are the clear winner among the starvation-free algorithms. Based on their implementations, we ported the following starvation-free algorithms from C99 to C11 and ran them in our microbenchmarks, using the same naming convention as on the survey by Buhr, Dice and Hesselink [2]:

- LamportBakery: Leslie Lamport's bakery algorithm [12];
- Hehner: An algorithm by Hehner and Shyamasundar [21];
- Szymanski: An algorithm with five different states by Boleslaw K. Szymanski [24];
- EisenbergMcGuire: An algorithm by Murray Eisenberg and Michael McGuire [9];
- PetersonBuhr: An algorithm devised by Peter Buhr, based on the tournament technique by Kessels [10], with Peterson's 2-thread algorithm at its basis;
- CRTurn: Our own algorithm with global spinning, using a turn variable, as described in section 2.1;
- CRHandover: Our own algorithm with local spinning, using an handover technique, as described in section 2.2;

In these implementations, we replaced the `Pause()` function with a `PAUSE` instruction when running on an x86 CPU, and replaced it with a call to `sched_yield()` for ppc and arm.

	Min Number release-stores	Min Number acquire-loads	Memory Usage
LamportBakery	$3+1$	$3N$	$2N$
Hehner	$2+1$	$2N$	N
Szymanski	$3+1$	$3N$	N
EisenbergMcGuire	$3+1$	$3(N+1)$	$N+1$
PetersonBuhr	$(2+1)log_2 N$	$log_2 N$	$5N+2Nlog_2 N$
CRTurn	$1+1$	N	$N+1$
CRHandover	$1+1$	$(N+4)$	$2N+1$

Table 2. Comparison table between the different algrithms for N-thread mutual exclusion. The first column shows the minimum number of sequentially consistent stores (stores with release) needed enter the critical section, plus the number needed to leave the critical section. The second column represents the minimum number of sequentially consistent loads (loads with acquire) used on the algorithm. The third column shows the memory usage in number of words, as a function of the number of threads N.

The source code for the algorithms and benchmarks used in this paper is available as part of the ConcurrencyFreaks library [23] under C11/papers/cralgorithm.

3.1 High contention microbenchmark

Figure 3 shows the results of the microbenchmarks when under a high contention scenario. The top two figures are for x86, with the left one for the AMD Opteron and the right one for the Intel Xeon. On the AMD Opteron, both PetersonBuhr and CRHandover stand out from the other starvation free algorithms. On Haswell, PetersonBuhr is clearly the best, with CRHandover being the second best. On ppc and arm, all starvation-free algorithms have similar performance.

3.2 Low contention microbenchmark

Having high contention on a lock is an undesired behavior because it implies a poor utilization of resources (all other threads are constantly waiting for the lock), causes a bottleneck on scalability (doesn't matter how many cores are in the machine running the application, only one core is doing work and all others are just spinning), and can cause severe latency delays (even with a starvation-free lock with linear wait, a thread may have to wait for $N-1$ context switches before it acquires the lock). This means that although the high contention scenario may be the most attractive scenario to study, we believe a low contention scenario to be much more realistic and, therefore, the most important to look at when choosing one of these algorithms as a lock implementation for a real case application.

We ran the same algorithms as for the high contention case and got the results shown in figure 4.

One detail on the plots shown in figures 3 and 4, is that the data point for one thread is not shown. This is due to the fact that if it is known a priori that there is only one thread attempting to enter the critical section, then there is no point in having a mutual exclusion lock protecting the critical section and, therefore, is makes no sense to show such a value.

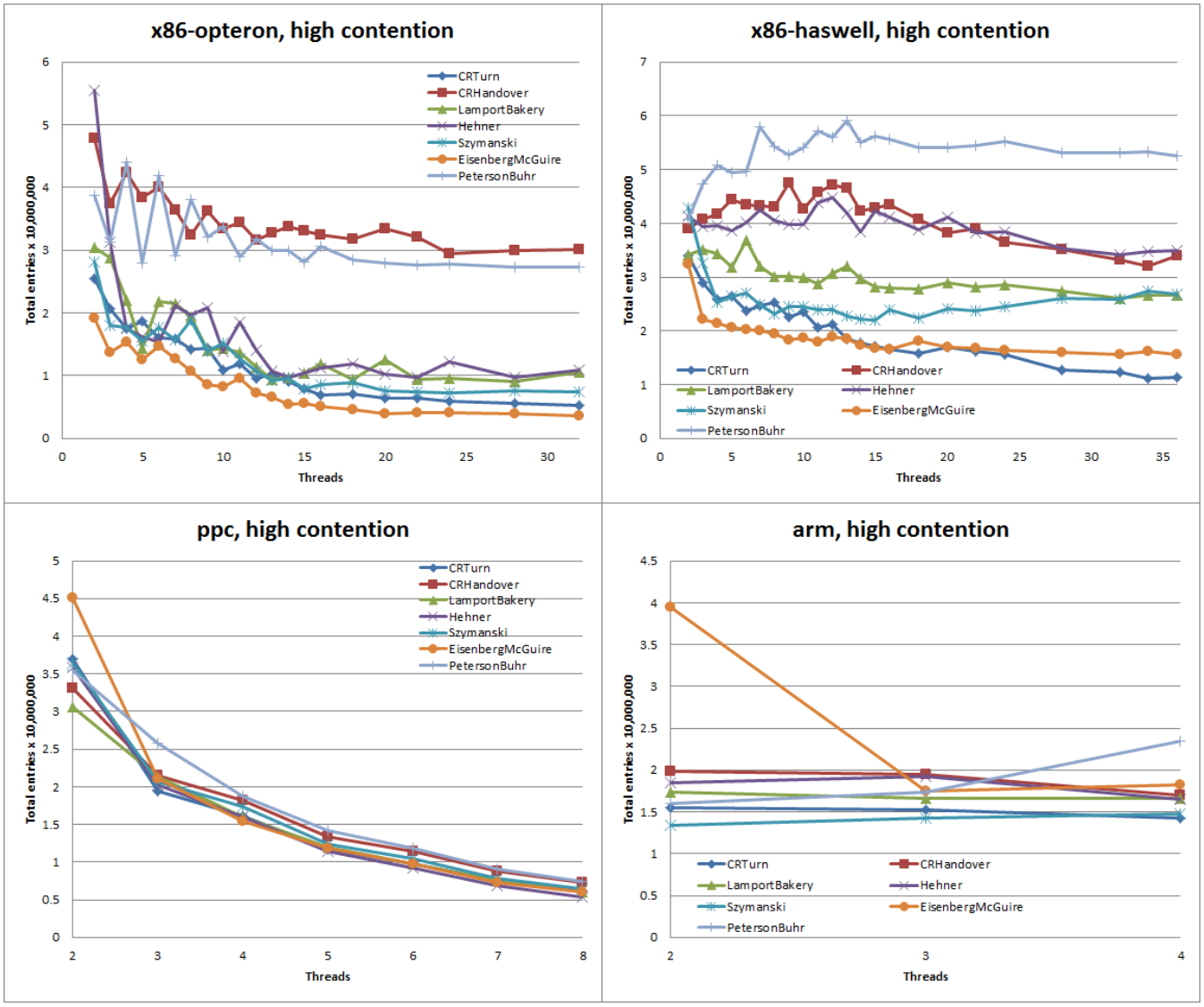

Figure 3. Plots for starvation-free algorithms under high contention

3.3 Analysis of the Results

Looking at the results in figure 3 for the high contention microbenchmark, we notice that across all architectures there are two algorithms that are consistently better than the others or among the best, they are PetersonBuhr and CRHandover.

The PetersonBuhr algorithm in the high contention scenario, after the critical section, it does $log_2(N)$ calls to `atomic_store()` to reset the states of each tournament, yet, the first call, the one at the root node, suffices to let the next waiting thread know that it is safe to enter the critical section, because the next waiting thread will be the `adversary` of the Peterson 2-thread algorithm on the root node of the tree of tournaments. If cache line padding is used, like we have on our implementation, then the next waiting thread will be spinning on two variables, the `turn` and the adversary's state, and it will be the only thread doing so. Ideally, it would be spinning on a single variable, but the penalty for spinning on two variables is small. We should point out that a given thread may spend a considerable amount of time fighting its way through the multiple tournaments until it reaches the root, and then again spend some time resetting the states back to their default value, but the time spent on these tasks is not relevant on the high contention case because if the thread were to reach the root sooner, it would simply go into spinning sooner and thus provide no visible benefit. However, we will show that on the low contention scenario this is not the case.

In the high contention scenario, the CRHandover algorithm will have the `handoverEnabled` variable always set to 1, which means that the next waiting thread is waiting on two variables, namely, `handoverEnabled` and its own entry in the `handover[]` array, as shown in lines 3 and 5 of Algorithm 3. The thread currently holding the lock, when completing its critical section, will do up to $(N-1)$ `atomic_load()` and then set the entry in `handover[]` corresponding to the next waiting thread from `NOTMINE` to `MYTURN`, thus effectively passing the lock with a single `atomic_store()`, as shown in line 16 of Algorithm 4.

There are two extra advantages of our two algorithms over PetersonBuhr. First, is that both CRTurn and CRHandover are starvation-free with linear wait (linear in the number of threads N),

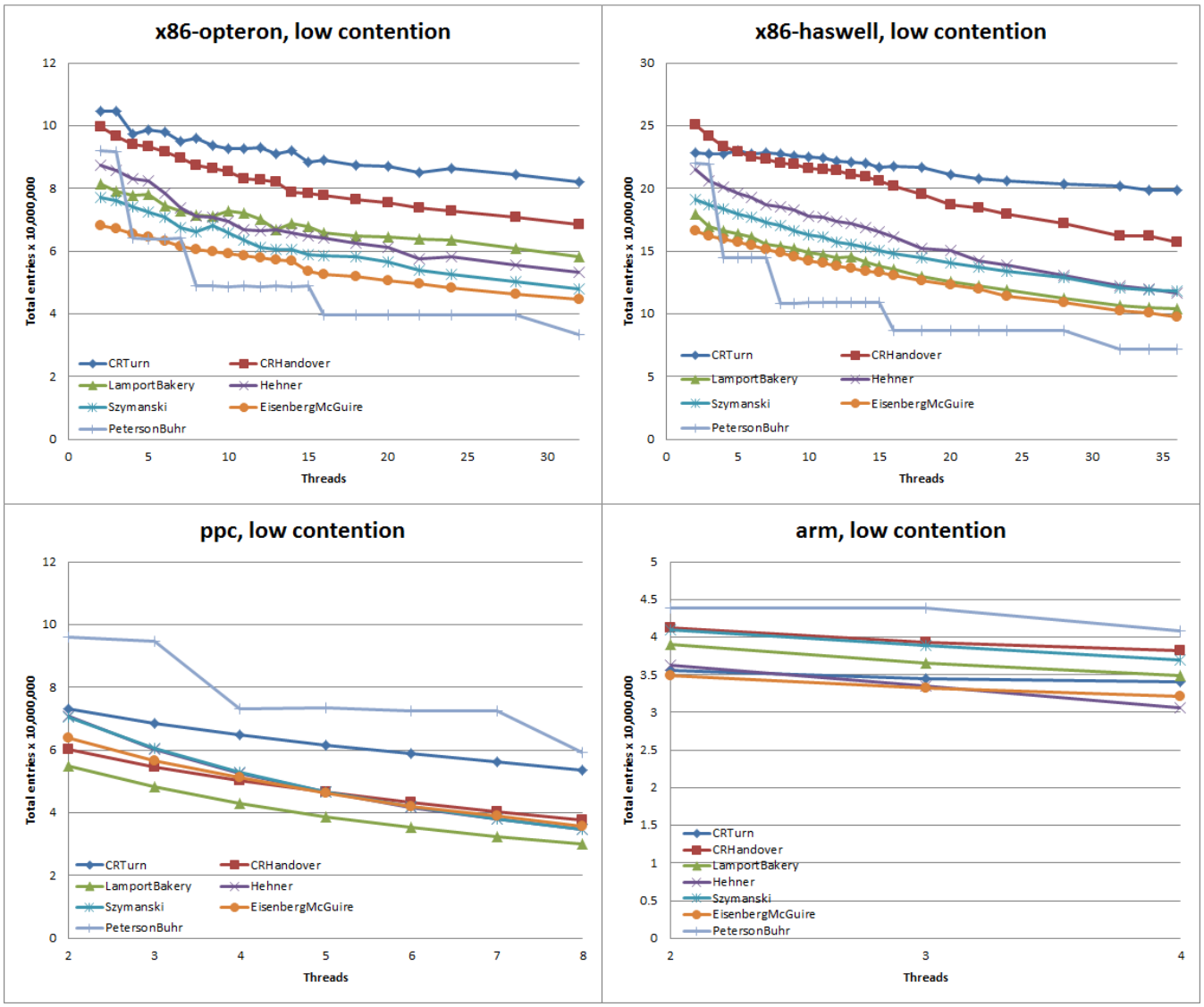

Figure 4. Plots for starvation-free algorithms under low contention

while PetersonBuhr has a wait that is theoretically unbounded. Second, is that the memory used by CRHandover is $2N + 1$ words and $N + 1$ for CRTurn, while for PetersonBuhr it is $5N + 2Nlog_2(N)$ words.

For the low contention scenario, from the starvation-free algorithms, the CRTurn and CRHandover are among the best across the different architectures, being surpassed only by PetersonBuhr on the ppc and arm architectures. This happens due to their low number of release-stores, just one to enter the critical section and another release-store after leaving the critical section, namely, lines 1 and 23 of Algorithm 1, and lines 13 and 33 of Algorithm 3. In a low contention scenario, the remaining stores done by these two algorithms are relaxed and therefore, have a smaller impact on performance.

An interesting angle to study, is the effects on the performance of the algorithm by providing starvation-freedom. Intuitively, we would expect that the more unfair an algorithm is, the higher potential for throughput it could provide, with the reasoning behind it being that, a very unfair algorithm will have just a few threads constantly getting the lock, or even always the same thread, thus reducing the number cache misses on the critical section and even cache misses on the variables of the algorithm itself.

Indeed, throughput-wise, the best algorithm across the multiple architectures is usually the LamportRetract, which happens to be the most unfair of all algorithms we considered, and in the high contention benchmarks has a tendency to give the lock always to the same one or two threads. In a certain sense, it is somewhat biased to compare algorithms with/without starvation-free guarantees, or even with different starvation-free guarantees.

Following such a logic, an engineer choosing one of the algorithms described in this paper to use in its application, should pick the one that is the most unfair possible, taking into consideration the particularities of the application, like the need or not for strong latency guarantees. If no fairness is needed and there are no important concerns about latency, then the best candidate is usually LamportRetract. On the other hand, if high fairness or strong guar-

antees of latency are needed, then PetersonBuhr is the best for ppc and arm, but CRHandover or CRTurn are the best for x86 and amongst the best candidates for the other architectures.

4. Conclusion

More than 50 years have passed since the question of whether or not mutual exclusion is possible in software was first posed. During this time, many algorithms have been proposed, with different properties, yet, we believe that our approach brings something new to the table, mostly due to its low number of stores in shared memory, while providing starvation-freedom with linear wait. Our two algorithms are the best, when compared with other starvation-free solutions in the low contention scenario, and in the high contention CRHandover is among the best, being only surpassed by PetersonBuhr, whose starvation-freedom is not linear and uses signficantly more memory.

A recent exaustive analysis [2] of the mutual exclusion algorithms using only loads and stores, has shown that tournament techniques provide good throughput with starvation-freedom. In conclusion, our algorithms provide similar or better throughput across multiple architectures, with even stronger latency guarantees and lower memory usage.

Acknowledgments

We would like to thank Dave Dice, Peter Buhr, and Wim Hesselink for making all of the source code used in their paper publicly available, which allowed us to make a faster progress on our own algorithms.

References

[1] Y. Bar-David and G. Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *Distributed Computing*, pages 136–150. Springer, 2003.

[2] P. A. Buhr, D. Dice, and W. H. Hesselink. High-performance n-thread software solutions for mutual exclusion. *Concurrency and Computation: Practice and Experience*, 2014.

[3] J. Burns and N. A. Lynch. Mutual exclusion using invisible reads and writes. In *In Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*. Citeseer, 1980.

[4] CPP-ISO-committee. C++ Memory Order. `http://en.cppreference.com/w/c/atomic/memory_order`, 2013.

[5] T. Craig. Building fifo and priorityqueuing spin locks from atomic swap. Technical report, Technical Report 93-02-02, University of Washington, Seattle, Washington, 1994.

[6] N. G. de Bruijn. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 10(3):137–138, 1967.

[7] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[8] E. W. Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*, pages 289–294. Springer, 2001.

[9] M. A. Eisenberg and M. R. McGuire. Further comments on dijkstra's concurrent programming control problem. *Communications of the ACM*, 15(11):999, 1972.

[10] J. L. W. Kessels. Arbitration without common modifiable variables. *Acta informatica*, 17(2):135–141, 1982.

[11] D. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966.

[12] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[13] L. Lamport. What it means for a concurrent program to satisfy a specification: why no one has specified priority. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 78–83. ACM, 1985.

[14] L. Lamport. The mutual exclusion problem: partiistatement and solutions. *Journal of the ACM (JACM)*, 33(2):327–348, 1986.

[15] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 165–171. IEEE, 1994.

[16] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[17] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 269–278. ACM, 1991.

[18] D. D. Peter Buhr and W. Hesselink. concurrent-locking. `https://github.com/pabuhr/concurrent-locking`, 2015.

[19] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[20] G. L. Peterson and M. J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 91–97. ACM, 1977.

[21] H. E. C. R. and R. K. Shyamasundar. An implementation of p and v, information processing letters. *Information Processing Letters*, 12(4): 196–197, 1981.

[22] P. Ramalhete and A. Correia. Tidex Lock. Technical report, 12 2014. URL `https://github.com/pramalhe/ConcurrencyFreaks/tree/master/papers/t`

[23] P. Ramalhete and A. Correia. Concurrency Freaks. `https://github.com/pramalhe/ConcurrencyFreaks`, 2015.

[24] B. K. Szymanski. A simple solution to lamport's concurrent programming problem with linear wait. In *Proceedings of the 2nd international conference on Supercomputing*, pages 621–626. ACM, 1988.