Simon König, 11702826
Viktor Beck, 11713110
Michael Ketter, 11701297

184.726 Advanced Multiprocessor Programming

TECHNISCHE UNIVERSITÄT WIEN

Project 2: Hardware supported Locks

1 Introduction

1.1 Description of the Project

One of the main challenges in multiprocessor programming is to make sure that shared datastructures are read from and written to in a sequential-like manner. A universal way how this can be achieved for arbitrary datastructures is to generate a lock which makes sure that only one thread can access at a time. Since reading and writing to a shared datastructure in a given algorithm is done quite frequently, it is important that the actual implementation of the used lock is as efficient and fast as possible. Therefore, implementations of such locks often utilize hardware supported operations to optimize the efficiency even further.

In this project, several implementations of locks using hardware supported operations should be implemented correctly. Furthermore, the performance and fairness parameter of the locks with varying thread number should be compared.

1.2 Implemented hardware-supported Locks

All the implemented locks were written in C. For the atomic data types and hardware-supported operations the library `<stdatomic.h>` was used. Due to consideration in terms of standardized usability, each lock implementation consists of the methods:

- `void init(Lock* self)`
- `void lock(Lock* self)`
- `void unlock(Lock* self)`
- `void destroy(Lock* self)`

and additionally, a data structure storing the lock's necessary information and an optional data structure storing the node properties for linked-list-like locks. As it can be seen, each method requires to be called at an object of a lock implementation. In theory, this enables the usage of several lock objects at once and therefore makes it possible to handle more than one shared data structure independent from each other. However, to simplify the testing procedure, this functionality is not considered in the analysis of this project. A short introduction and discussion of the implementation of of each lock is provided in the following.

1.2.1 Test-and-set Lock

The lock mechanism is simply changing an atomic flag from true to false when acquiring the lock. Unlocking is achieved by setting the flag to true again. Since only an atomic boolean variable is stored, sophisticated memory management is not necessary, which makes the implementation fairly simple as it can be seen in code snippet 1.

Listing 1: Test-and-set Lock

```
typedef struct Lock {
atomic_bool lock_flag;
} Lock;

void lock(Lock* self) {
    while (atomic_exchange_explicit(&self->lock_flag, 1, memory_order_acquire)){};
}

void unlock(Lock* self) {
    atomic_store_explicit(&self->lock_flag, 0, memory_order_release);
}
```

Due to the simple implementation the performance of the lock is quite good at the first sight. For a higher number of threads this is certainly not true, since one `atomic_exchange_explicit(...)` call invalidates all the threads cached copies of the boolean variable. This initiates bus transactions which puts high contention on the memory interconnect. Additionally, the lock is highly unfair and a given thread can be overtaken an unlimited number of times.

1.2.2 Test-and-test-and-set Lock

The Test-and-test-and-set lock is generally identical to the Test-and-set lock, with the small change that an additional test of the atomic bool variable is added. The implementation which is used in our project is shown in code snippet 2.

Listing 2: Test-and-test-and-set Lock

```
typedef struct Lock {
atomic_bool lock_flag;
} Lock;

void lock(Lock* self) {
    while (1) {
        while (atomic_load_explicit(&self->lock_flag, memory_order_relaxed)) {}
        if (!atomic_exchange_explicit(&self->lock_flag, 1, memory_order_acquire)) {
            return;
        }
    }
}

void unlock(Lock* self) {
    atomic_store_explicit(&self->lock_flag, 0, memory_order_release);
}
```

In general, the hardware-supported function `atomic_load_explicit(...)` executes with less overhead than the `atomic_exchange_explicit(...)`. Since most of the threads will get stuck in the loop with the former they will not execute the latter as often as in the Test-and-set lock which leads to less bus transactions when threads are trying to acquire the lock and thus to better scaling

performance. However, the adaption of the implementation with respect to the Test-and-set lock does not change anything in its highly unfair behaviour. Theoretically, a given thread could still be overtaken an unbounded number of times.

1.2.3 Ticket Lock

The idea of the ticket lock is, that it works like a ticket dispenser where the customers have to wait until their ticket is called. The implementation used in this project is shown in code snippet 3.

Listing 3: Ticket Lock

```
typedef struct Lock {
    atomic_int ticket;
    atomic_int served;
} Lock;

void lock (Lock* self) {
    int my_ticket = atomic_fetch_add(&self->ticket, 1);
    while (atomic_load(&self->served) != my_ticket);
}

void unlock(Lock* self) {
    atomic_fetch_add(&self->served, 1);
}
```

A thread trying to acquire the lock performs `atomic_fetch_add(...)`. Hereby, the executing thread stores the number saved in the lock data structure and increases this number by one. This happens in an atomic manner and therefore only one thread at a time can fetch and add the locks ticket number. A second number is stored in the locks data structure, which is initialized to zero. The thread with ticket number equaling the second number stored in the locks data structure is able to acquire the lock. All the other threads spin in the while loop since their ticket is currently not the active one. Unlock will atomically increase the variable `served` by 1 and the next thread can enter the critical section. The Ticket lock ensures fairness because the threads will acquire the lock in the order in which they retrieved their ticket numbers.

One problem with this lock is that the ticket number grows unbounded. This problem may not be relevant in our toy model scenarios, but can lead to problems when using the lock unconsciously. Additionally, the lock is not fault tolerant meaning that if one thread stalls or crashes, all the other threads cannot continue as well.

1.2.4 Array Lock

The array locks uses a circular array of atomic boolean values with one element per thread. This data structure represents the list of which thread is allowed to acquire the lock and which threads have to wait. In our implementation, which is depicted in the code snippet 4, there is an additional atomic integer acting as ticket dispenser for threads trying to acquire the lock.

Listing 4: Array Lock

```
#define SIZE omp_get_max_threads()

typedef struct Lock {
  bool* flags;
  _Atomic int tail;
```

```
} Lock;

__thread int mySlot;

void lock(Lock* self) {
    mySlot = atomic_fetch_add(&self->tail, 1) % SIZE;
    while (!self->flags[mySlot]) {
    }
}

void unlock(Lock* self) {
    self->flags[mySlot] = false;
    self->flags[(mySlot + 1) %
    SIZE] = true;
}
```

A thread trying to acquire the lock gets a ticket which determines the threads position in the boolean array. While this dedicated array position is set to false, the thread spins in the while-loop. If a thread is currently in the critical section and calls the unlocking method, it sets its own array element to false and the following array element to true. The thread associated with the following array element is therefore allowed to enter the critical section.

In general, this lock has similar properties as the ticket lock - especially sharing the strong fairness property. Additionally, this lock fixes the problem of the ticket lock that there exists an unbounded integer variable. Admittedly due to reasons of simplicity, this is not the case in our implementation of the array lock. However, in exchange to having an unbound integer value, the array lock stores an array of boolean values of size, which is determined by the number of threads. The lock is therefore not space efficient anymore. Another negative property of the array lock - similar to the ticket lock - is that it is not fault tolerant, which can lead to problems when using the lock in real-world applications.

1.2.5 CLH Lock

The following three locks - CLH, MCS and Hemlock - can be categorized into a linked list approach. This approach has the advantage that the nodes of the list are stored in separate memory positions, which reduces false sharing and therefore prevents accidental errors. The CLH lock was independently introduced by Craig and Landing [1] and Hagerstin [3] - explaining the name of the lock. The implementation of the CLH lock used in this project can be seen in code snippet 5.

Listing 5: CLH Lock

```
typedef struct Node
{
    _Atomic bool succ_must_wait;
    struct Node* next;
    char padding[64];
} Node;

static __thread struct Node* MyNode;

typedef struct
{
    Node dummy;
    _Atomic (Node *) tail;
} Lock;
```

```
15
16 void lock(Lock * self)
17 {
18     MyNode = (Node*)malloc(sizeof(struct Node));
19     atomic_store(&MyNode->succ_must_wait, 1);
20     MyNode->next = (Node*) atomic_exchange_explicit(&self->tail, MyNode, memory_order_seq_cst);
21     while (atomic_load(&MyNode->next->succ_must_wait)) {}
22 }
23
24 void unlock(Lock * self)
25 {
26     atomic_store(&MyNode->succ_must_wait, 0);
27 }
```

Since the lock has a linked list approach, two different data structures are needed implementationwise. The data structure `Node` represents the array elements in the linked list. Each thread initializes and stores its own object of `Node`. When a thread creates a new node, it is appended at the end of the linked list. Additionally, a pointer to its precursor and an atomic boolean value, which specifies if a potential successor has to wait. The data structure `Lock` stores the data needed for the actual lock. It contains a dummy node, which represents first empty sentinel node and a pointer to the current last node of the linked list.

When a thread calls the lock function, a new member of `Node` is created and its atomic boolean value is set to 1. Subsequently, `atomic_exchange_explicit(...)` is used to assign the previous tail pointer as next pointer and let the tail point to the newly created node. While the node pointed to by the next pointer has set its atomic boolean value to 1, the new node has to spin and wait. The unlocking function only consists of changing the atomic boolean value from 0 to 1. Theoretically, the unlocked node is not needed anymore and could be deleted. Therefore only as many nodes as currently wait in the lock need to be stored. Due to simplicity reasons, this was not carried out in this projects implementation.

Similar as the previous locks, the fairness property is quite strong, making the lock intolerant with respect to faults. Other than that, the linked list approach increases the scaling of the performance considerably compared to the previous Test-and-set and Test-and-test-and-set locks.

1.2.6 MCS Lock

Compared to the CLH lock, the MCS lock maintains the head of the lock instead of the tail. This implies that new nodes are appended at the beginning of the linked list. Similar to the CLH lock, the memory positions of the individual nodes are separated from each other meaning that false sharing is avoided. It was introduced for the first time by Mellor-Crummey and Scott [4]. The implemention used in this project is depicted in code snippet 6.

Listing 6: MCS Lock

```
1 typedef struct node{
2    _Atomic bool locked;
3    struct node* next;
4    char padding[64];
5 } node;
6
7 typedef struct Lock{
8    _Atomic (struct node*) head;
```

```
} Lock;

static __thread struct node mynode = {0, (struct node*) NULL, ""};

void lock(struct Lock* self){
    struct node* n = &mynode;
    atomic_store(&n->next, (struct node*) NULL);
    struct node* pred = atomic_exchange(&self->head, n);
    if (pred != (struct node*) NULL) {
        atomic_store(&n->locked, 1);
        pred->next = n;
        while (atomic_load(&n->locked));
    }
}

void unlock(struct Lock* self)
{
    struct node* n = &mynode;
    if (n->next == (struct node*) NULL){
        if (atomic_compare_exchange_strong(&self->head, &n, (struct node*) NULL)) {
            return;
        }
        n = &mynode;
        while (n->next == (struct node*) NULL);
    }
    atomic_store(&n->next->locked, 0);
    n->next = (struct node*) NULL;
}
```

In the MCS lock, every thread has its own `Node` instance saved as local static variable. Each node instance stores - similar as the CLH lock - a pointer to the next element in the linked list and an atomic boolean value, representing the current status of the lock. The data structure representing the actual lock only stores an atomic pointer to the head of the linked list.

When a thread calls the lock method, the locally saved node is reset to starting state. Afterwards, the node is inserted as new first node by using an `atomic_exchange(...)` operation. The head pointer of the lock as well as the next pointer of the previous head node now point to the inserted node. Additionally, the atomic boolean variable of the new node is set to 1. If no thread has tried to acquire the lock before, the current node is assigned as head node and can go straight to the critical section. In the case of an unlock call, firstly it is verified if there is another thread waiting to acquire the lock. If this is the case, then the atomic boolean variable of the current node is set to 0, implying that the next thread can enter. Additionally, the pointer to the next node is invalidated.

The properties of the MCS lock are quite similar to that of the CLH lock. Nevertheless, one slight difference is that the unlock method is in general not wait free anymore since it has to be waited for the next lock owner to set the next pointer. Regarding memory requirements it is slightly more expensive than the CLH lock since it has to store a node object for every thread the whole time.

1.2.7 Hemlock

The Hemlock was developed quite recently in 2022 by Dice and Dogan [2]. In general it utilizes a linked list approach with minimal memory requirements compared to the MCS and CLH lock.

Because of this approach, it claims to still spin locally in most times. The Hemlock implementation used in this project can be found in codesnippet 7.

Listing 7: Hemlock

```
typedef struct node{
   _Atomic struct Lock* Grant;
} node;

typedef struct Lock{
   _Atomic struct node* tail;
} Lock;

__thread struct node mynode = {(_Atomic struct Lock*) NULL};

void lock(struct Lock* self){
    struct node* n = &mynode;
    atomic_store(&n->Grant, (_Atomic struct Lock*) NULL);
    struct node* pred = (struct node*) atomic_exchange(&self->tail, (_Atomic struct node*) n);
    if (pred != (struct node*) NULL) {
        while (__sync_val_compare_and_swap(&pred->Grant, self, (_Atomic struct Lock*) NULL) != (
    _Atomic struct Lock*) self) {};
        atomic_store(&pred->Grant, (_Atomic struct Lock*) NULL);
    }
}

void unlock(struct Lock* self)
{
    struct node* n = &mynode;
    struct node* pred = (struct node*) __sync_val_compare_and_swap(&self->tail, (_Atomic struct
    node*) n, (_Atomic struct node*) NULL);
    if (pred != n){
        atomic_store(&n->Grant, (_Atomic struct Lock*) self);
        while(atomic_load(&n->Grant) != (_Atomic struct Lock*) NULL);
    }
}
```

The implementation uses two different data structures representing the lock and the nodes. A node object stores a pointer to its corresponding lock and a lock object stores a pointer to the tail of the linked list. Each thread has its own `Node` entity initialized and stored locally.

When the lock method is called, the active node reinitializes at first. Afterwards, it appends itself at the end of the linked list using an `atomic_exchange(...)` operation. While the previous node does not point to the lock, the current node spins and waits. If the while loop evaluates to false, meaning the previous node points to the lock, the pointer of the previous node is invalidated by its successor before entering the critical section. In the unlock method, it is at first checked if there is a successor node which wants to acquire the lock. If that is the case, the nodes pointer is set to point to the lock. Afterwards, the node waits until its successor enters the critical section and invalidates the pointer again.

As the previous locks, the Hemlock has a strong fairness property. Due to the linked list approach, it is ensured that the waiting happens locally most of the times. This implies good performance scaling with higher number of threads. Since for every additional thread, only a single pointer has to be stored, the implementation is lightweighted in therms of memory requirements. In comparison to the CLH lock, the unlock method of the Hemlock is not wait-free anymore, since the successing

thread has to edit the predecessor pointer.

1.2.8 Baseline Lock

To analyze the performance of a lock it is necessary to compare against a certain baseline. The lock, which was used for that is the standard lock implementation of the openMP library `<omp.h>`. To adapt the useability to our framework, the lock was wrapped in the previously introduced structure as it can be seen in codesnippet 8.

Listing 8: Baseline Lock

```
#include <omp.h>

typedef omp_lock_t Lock;

void init(Lock* self){
    omp_init_lock(self);
}

void lock(Lock* self) {
    omp_set_lock(self);
}

void unlock(Lock* self) {
    omp_unset_lock(self);
}

void destroy(Lock* self){
   omp_destroy_lock(self);
}
```

1.2.9 Incorrect Lock

As a baseline for the correctness test, an incorrect lock which allows that an unbound number of threads enter the critical section was implemented as well. The lock was merely used for counter-testing the correctness test and is designed in a way that it fails the correctness test. The implementation of this lock can be seen in code snippet 9.

Listing 9: Incorrect Lock

```
typedef struct Lock {} Lock;

void init(Lock* self) {}

void lock(Lock* self) {}

void unlock(Lock* self) {}

void destroy(Lock* self) {}
```

2 Considerations to test for Correctness

In the following, an approach is described how correctness can be tested in an applied context. To test for correctness an array filled with random numbers in parallel is summed and compared to the sequentially computed sum. The summation is done in the critical section of the tested lock. In the critical section, the non-atomic index representing the currently accessed array element is increased by one every iteration. This is done after the current element is added to the sum. If the lock would not satisfy mutual exclusion it would eventually lead to different sums for the sequential and the parallel summation, since multiple threads could hold the same index and add an identical value multiple times in a row.

When applying this concept on the implemented locks, all of them passed the test. Counter-testing with an incorrect lock led to a difference in the sums and it did therefore not pass the test. Hereby, we made sure that the implemented locks work as planned and that it is not the implementation of the correctness test itself that ensures mutual exclusion. Although, this is not a universal proof for mutual exclusion for the locks since we cannot rule out the possibility that a certain case is not covered. However, since the implemented correctness test acts as realistic toy model for an actual use-case of the locks, it can be concluded that the lock implementation should yield correct results for calculating arbitrary mathematical operation in parallel.

3 Considerations for the Benchmarks

3.1 Throughput

Throughput can generally be interpreted as total number of executed operations inside the critical section per unit timestep. The number of acquisitions strongly depends on the locks contention e.g. how many threads want to access the critical section at the same time. To analyze the behaviour with low contention as well as high contention, following considerations were taken into account. High contention is generally reached when all threads want to acquire the lock at the same time. This is reached when the main parallel workload is done in the critical section. Low contention, on the other hand, is reached, if a substantial workload has to be executed outside of the critical section.

Inspired by the benchmarks done by Dice and Dogan [2] this workload is represented by simply delaying each thread randomly before trying to acquire the lock. As it can be seen in codesnippet 10, the random delay is done utilizing meaningless random operations. For the high contention benchmark, this random delay is omitted. In the critical section, a small workload consisting of five random operations is done in both benchmark versions.

Listing 10: Cutout of the random delay in low contention benchmark

```
unsigned int localSeed = time(NULL) + omp_get_thread_num();
srand(localSeed);
unsigned int localSteps = rand_r(&localSeed) % 20;
for (unsigned int j = 0; j < localSteps; ++j) {
  rand_r(&localSeed);
}
```

After the timed parallel region was executed succesfully, the throughput was calculated. The total number of lock acquisation was designed to be an input parameter and is therefore as customizable as needed. Additionally, the number of repetitions of the experiment can be specified via input

parameter as well. Finally, the mean throughput and a standard deviation of all experiments is calculated.

3.2 Latency

The latency of a lock is the time it takes a thread to acquire a lock. It measures the delay introduced by the locking mechanism when entering the critical section. For the conducted latency measurement, similarily to the throughput low contention measurement, a dummy working section was introduced ahead of the critical section. Thus decreasing contention and allowing lower thread latencys. The experiment was repeated a predefined number of times before the mean latency and the corresponding standard deviation were calculated. The total number of lock acquisitions and the repetitions of the experiment were again defined as input parameter.

In general, low latencys, in the order of 50-100 ns, are expected. On our local machines the latency for all locks are below or around 100 ns. The latencies on the multi-core nebula system are generally higher by a factor of 2. This may be due to lower clock cyles on the nebula system. For comparison, the clock cycle on the nebula system is around 1200 MHz, while the clock cycle at our private mac OS Macbook is 2300 MHz. This difference would explain the increase in latency by a factor of 2.

3.3 Fairness

To benchmark the fairness of the locks it is simply checked how often each thread acquires the lock for a fixed predefined number of total lock acquisitions. In code snippet 11 it can be seen that the threads run in the loop until the lock was acquired a certain number of times. Each thread increases its own counter inside the critical section after acquiring the lock. This was repeated for a predefined number of times to produce statistically valid results. In the end, an array of counters was yielded where it can be seen how often each thread acquired the lock.

Listing 11: Cutout of fairness.c

```
1     for (int iter = 0; iter < reps; iter++){
2
3         #pragma omp parallel shared(LOCK, counter, lock_calls)
4         {
5             while(atomic_load(&lock_calls) < max_lock_calls){
6
7
8             // add 1 to lock_calls
9             atomic_fetch_add(&lock_calls, 1);
10
11            // Acquire the lock
12            lock(&LOCK);
13
14            // Thread is in critical section - write to counter array
15            counter[omp_get_thread_num()] += 1.0;
16
17            // Release the lock
18            unlock(&LOCK);
19           }
20        }
21
22        // reset lock_calls
23        atomic_store(&lock_calls, 0);
```

```
24     }
```

The total number of lock acquisitions and repetitions of the experiment were designed as input parameter and are therefore individually customizable. In general, it is expected that a fair lock will produce similar values for each counter while an unfair lock will have strongly varying counters.

4 Benchmark results and Discussion

4.1 Throughput

The benchmarking experiment was executed with the thread numbers 1,2,3,4,5,8,10,16,32,50 and 64. The total lock acquisition number was set to 1000 and each experiment was repeated 50 times.

First take a look at throughput with high contention. In figure 1 one can see the behaviour of of all seven locks implemented in the project. The *BaselineLock* is the built-in OpenMp lock. All locks show a similar behaviour. They perform best for one thread and then throughput drops off (noted that the y-axis is in logarithmic scale). This behaviour is expected in the high contention case and is due to multiple threads spinning in the lock mechanism and having to wait before they can acquire the lock. Because almost all threads try to acquire the lock at the same time, threads stall while waiting ahead of the critical section. Traffic at the lock acquisition step leads to throughput and hence performance drop off. In conclusion one can say, that if it is expected that a critical section is highly contended, there is little gain by installing multiple threads. The performance even decreases with higher thread counts.

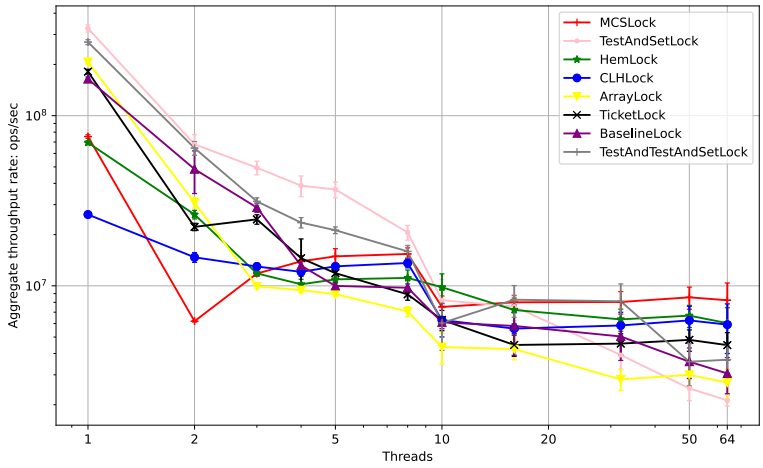


Figure 1: Throughput High Contention

Second take a look at the throughput with low contention. In figure 2 one can compare all locks implemented in the project with the built in OpenMp lock *BaselineLock*. The plot matches the expectations that all locks increase their performance with increasing thread count - until thread count of 6-8. From there on, one would expect the more sophisticated locks - like Hemlock, CLH lock and MCS lock - to outperform the simpler lock mechanism - like Test-And-Set or Ticket-Lock. However, this is not the case, since all locks show similar behaviour. The performance peaks around

16 spawned threads, which goes along our expectations. For one thread there is a positive throughput outlier: the CLH-Lock.

Compared to the high contention case one observes the necessity for multiple threads. The throughput rate increases with the thread count until it peaks at around 16 threads, hence low contention/heavy workload code sections prove to be a useful field for multi-thread environments.

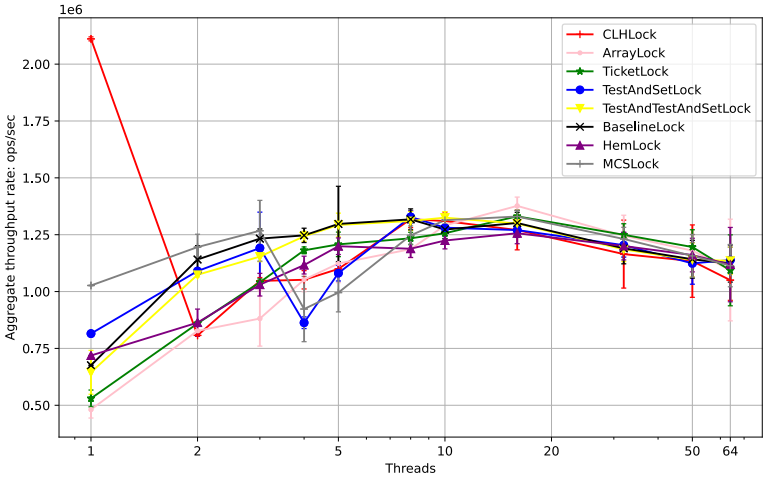


Figure 2: Throughput Low Contention

4.2 Latency

The latency for 4, 16 and 32 threads is shown in the figures below. Compared to the above mentioned expectations for latency, the measurement do not fully match the expected values, since we were expecting around 30-50 ns lock acquisition latency. In the figures for 4 threads (figure 3) and 16 threads (figure 4) the latency for the CLH lock is about 600 ns and has a high standard deviation. All other locks have a latency of at most half of the CLH lock.

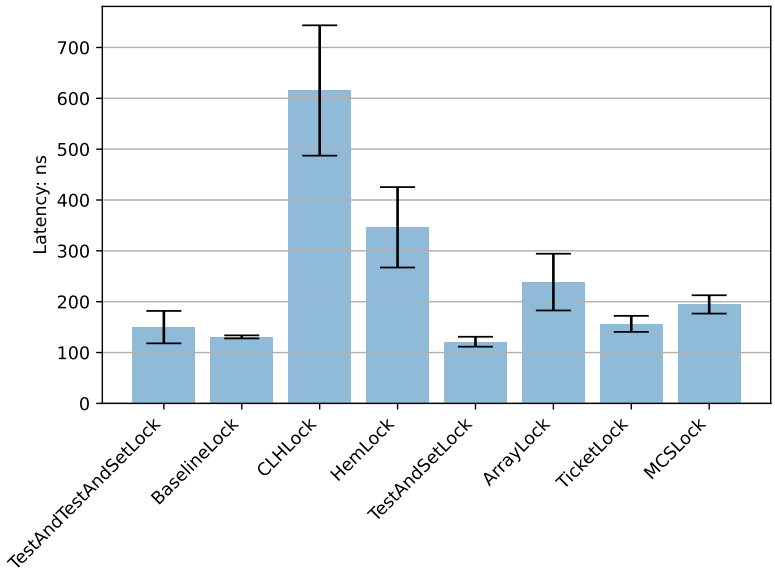


Figure 3: Latency: 4 threads

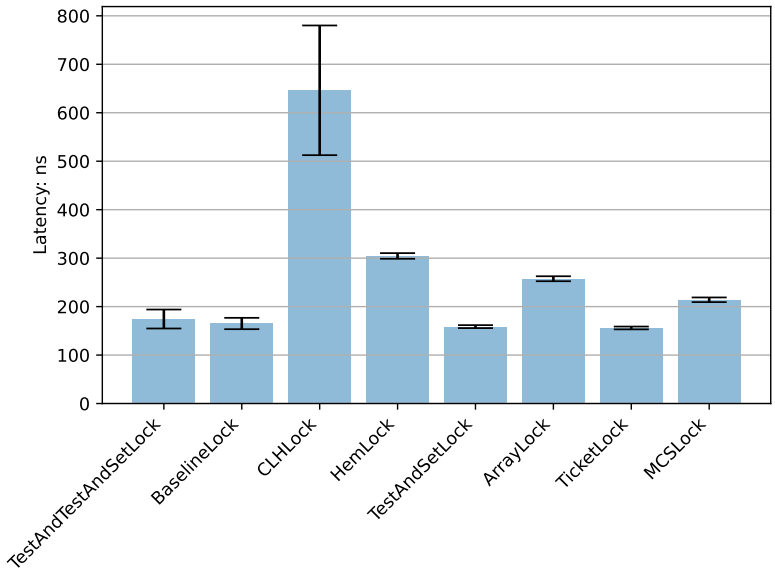


Figure 4: Latency: 16 threads

Looking at the 32 thread case in figure 5, almost all lock latency measures are in the range of microseconds. To analyse this one has to take lock contention into account. With increasing thread count and constant total number of lock acquisitions, lock contention increases automatically. Using the same reasoning as in the throughput case we can state, that high contention leads to long waits i.e. high latency. This is the underlying reason for overall lower throughput in the low contention case than in the high contention case.

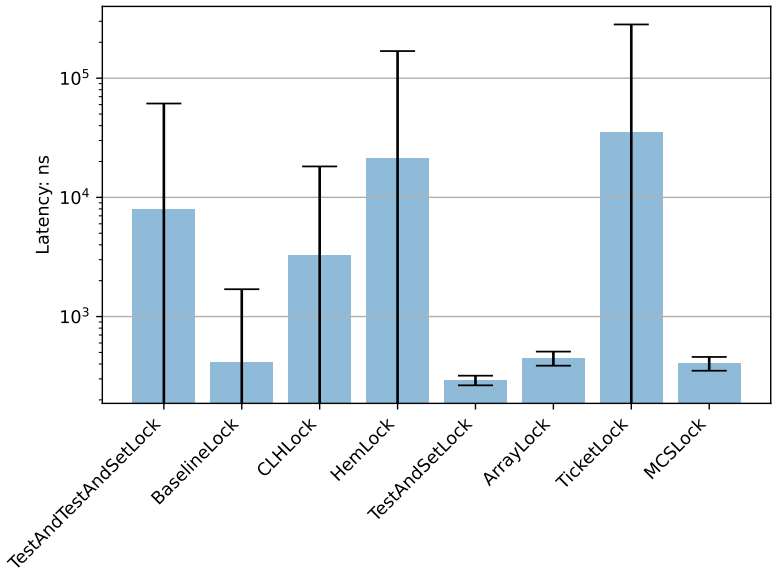
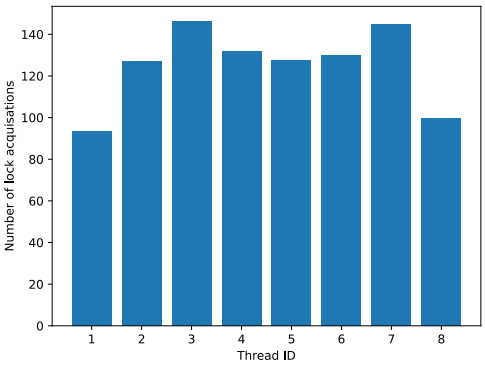


Figure 5: Latency: 32 threads

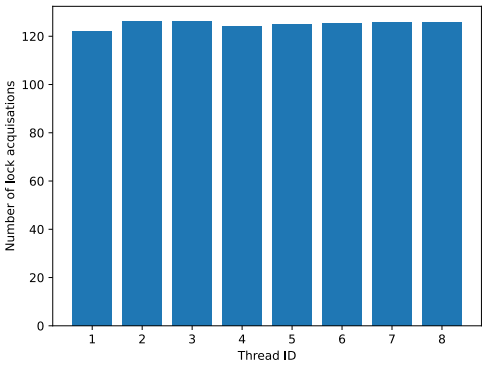
4.3 Fairness

Figure 7 shows the lock acquisition distribution of all implemented locks for utilizing 8 threads. The total number of lock acquisitions was set to 1000, implying that every thread must enter the critical section for 125 times if the lock is perfectly fair. As it can be seen in the figure, all locks beside the test-and-set lock, the test-and-test-and-set lock and the baseline lock fulfill this criterion quite perfectly. This results agrees with the theoretical observations since only the test-and-set lock and the test-and-test-and-set lock are considered unfair locks. Surprisingly, the baseline lock seems to be unfair as well. Since the baseline lock represents the standard lock of the `openMP` library, it is surprising that an unfair lock is the method of choice.

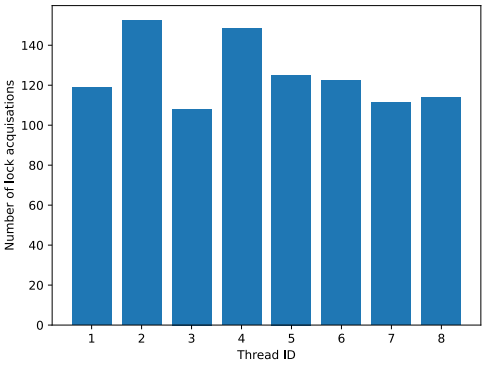
To investigate how an increasing number of threads influences the fairness property, previously described test was repeated with 64 threads. All other test parameter were kept unchanged. The resulting distribution plot of all implemented locks is shown in figure 9. In the plots it can be seen, that the qualitative behaviour of the locks does not change at all - meaning the three unfair locks are still unfair while all the other locks are fair. However, the distribution of the fair locks seem to be more fluctuating than previously. This is probably due to the fact that 1000 is not divisible through 64 without getting a remainder. Therefore, some threads always acquire the lock at least once more than the lock with minimal acquisition. The only lock, where this explanation does not seem to fit is the Hemlock. However, we could not come up with a reasonable explanation for the behaviour of this lock.



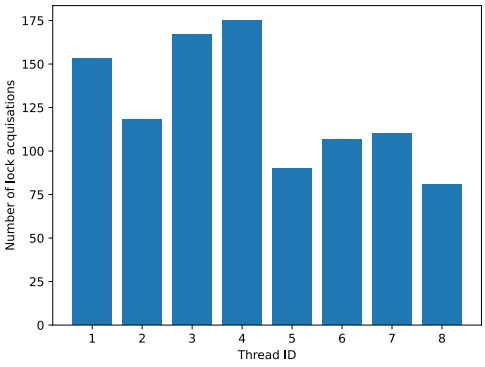
Baseline lock



Ticket lock



Test-And-Set lock



Test-And-Test-And-Set lock

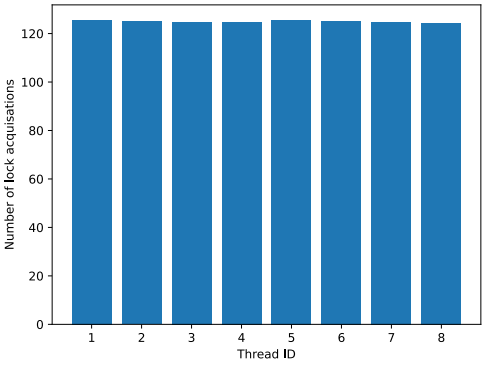
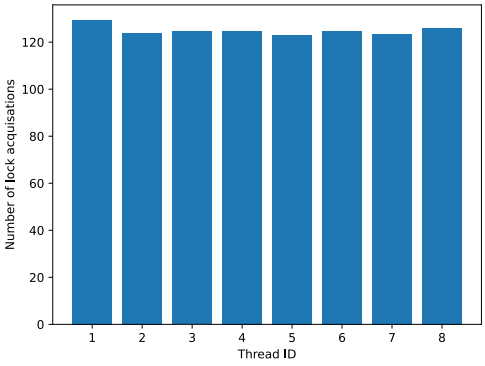
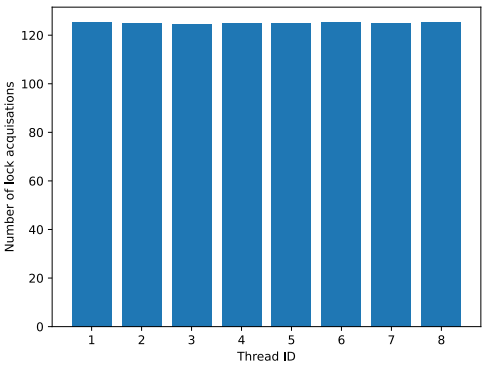


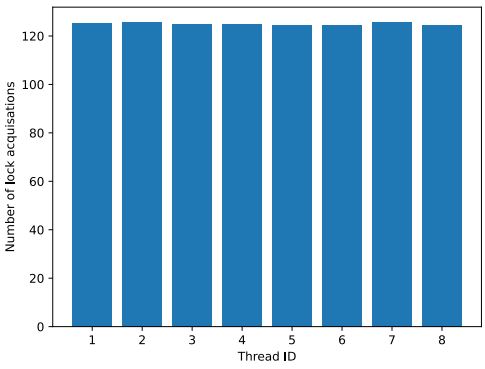
Figure 6: Array lock



CLH lock

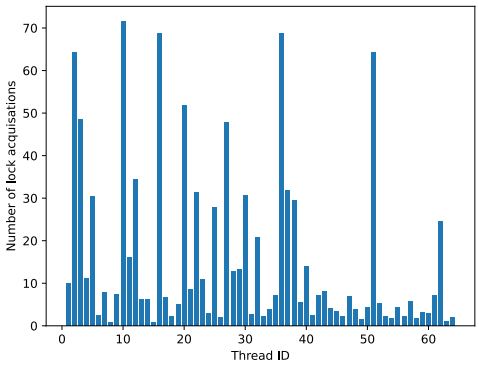


MCS lock

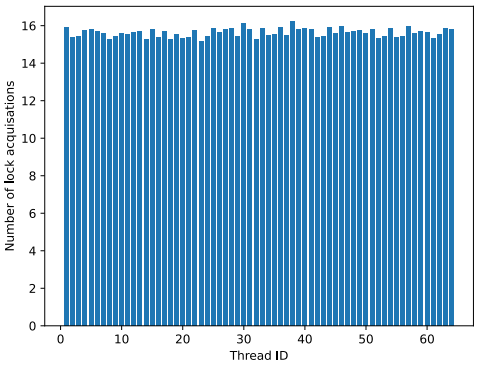


Hemlock

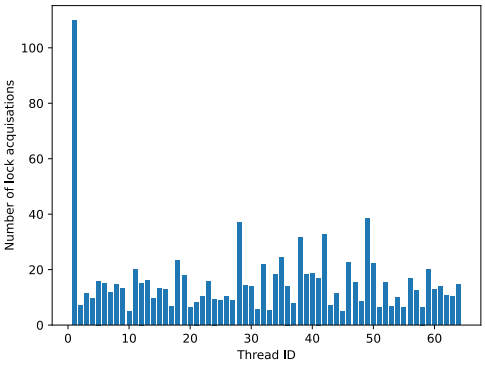
Figure 7: Comparison of the fairness for 8 threads



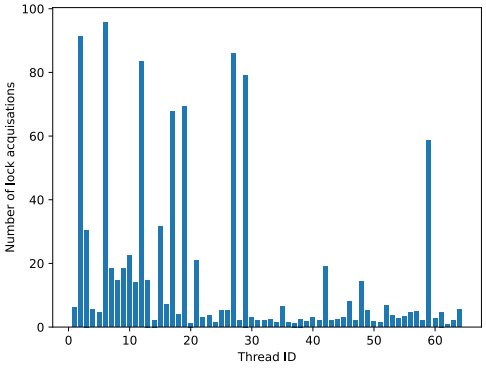
Baseline lock



Ticket lock



Test-And-Set lock



Test-And-Test-And-Set lock

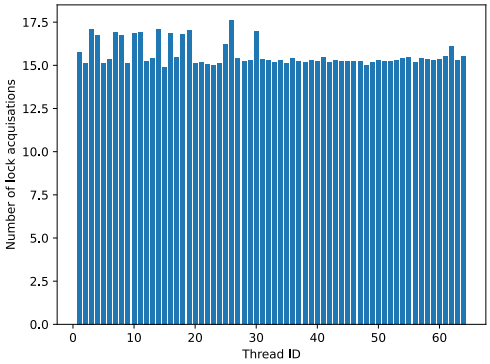
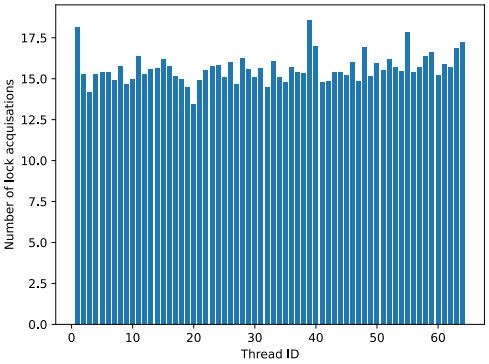
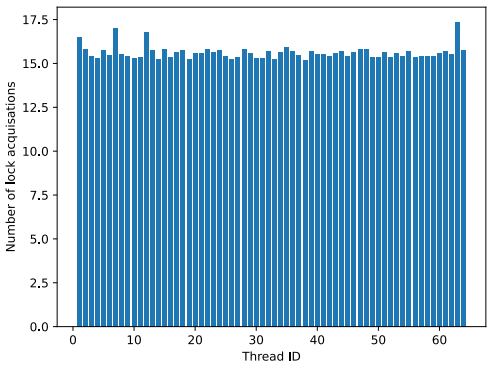


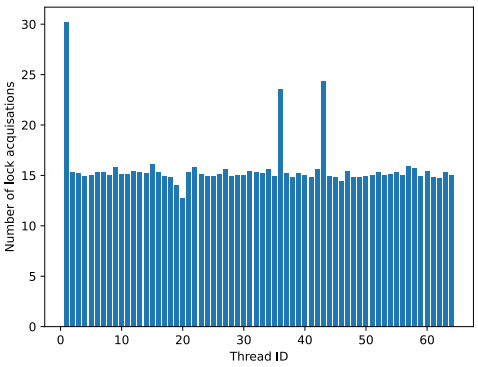
Figure 8: Array lock



CLH lock



MCS lock



Hemlock

Figure 9: Comparison of the fairness for 64 threads

5 Literature

[1] Travis S. Craig. “Building FIFO and Priority-Queuing Spin Locks from Atomic Swap”. In: (1993).

[2] Dave Dice and Alex Kogan. “Hemlock : Compact and Scalable Mutual Exclusion”. In: (2022). arXiv: 2102.03863.

[3] Peter S. Magnusson, Anders Landin, and Erik Hagersten. “Queue Locks on Cache Coherent Multiprocessors”. In: (1994), pp. 165–171.

[4] John M. Mellor-Crummey and Michael L. Scott. “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors”. In: *ACM Trans. Comput. Syst.* 9.1 (Feb. 1991), pp. 21–65. ISSN: 0734-2071. DOI: 10.1145/103727.103729. URL: https://doi.org/10.1145/103727.103729.

Technical Details of the Implementation

The GitHub repository can be found under `https://github.com/simon-koenig/Hardware-Locks`.

Requirements

To run all the scripts of the project one needs to have the following dependencies installed:

- GCC
- Python3
- Numpy
- Matplotlib
- Pandas

For more information please take a look at the README file.

Used system

2 AMD EPYC 7351P 32 core processors, 2 way hyperthreading , 1.2GHz, total 64 cores, 256GB main memory
Kernel: Debian GNU/Linux 10 (buster)
Software specification: gcc (Debian 8.3.0-6) 8.3.0, Python 3.7.3