184.726 Weiterführende Multiprocessor Programmierung

Programming Project Report

Benjamin Gruber (11770987) — Dominik Freinberger (11708140)

June 15, 2023

Register Locks

"Too many locks, not enough keys." Sarah Dessen

Contents

References

[BW90] Kenneth Block and Tai-Kuo Woo. "A more efficient generalization of Peterson's mutual exclusion algorithm". In: *Information Processing Letters* 35.5 (1990), pp. 219–222. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(90)90048-3. URL: https://www.sciencedirect.com/science/article/pii/0020019090900483.

[Ala05] K. Alagarsamy. "A mutual exclusion algorithm with optimally bounded bypasses". In: *Information Processing Letters* 96.1 (2005), pp. 36–40. ISSN: 0020-0190. DOI: https://doi.org/10.1016/j.ipl.2005.05.015. URL: https://www.sciencedirect.com/science/article/pii/S0020019005001420.

[Trä22] Jesper Larsson Träff. *AMP Mutual Exclusion Slides*. 2022.

1 Benchmarking Suite

Prior to implementation details and discussion of the different locks' characteristics, we need to provide a sound benchmarking set-up that allows to evaluate the lock implementations' behaviour with respect to fairness, latency and throughput. In the following we will present the experimental set-up and all related considerations in this regard.

1.1 Fairness

According to the Cambridge Dictionary 'fairness' is defined as the characteristic of a system or a human to treat all it's subjects equally or in a way that is right or reasonable. In the context of n-thread locks we can fall back on the intuition provided in [Trä22]: We "would like that if A calls *lock()*; before B then B cannot overtake A, that is B cannot enter the critical section before A". Generalized for *n* threads: Overtaking should be kept within a reasonable bound.

1.1.1 Measuring Fairness

As a result, the basic property of interest for is the number of threads *j* obtaining access to the critical section inbetween **two** accesses by thread *i*. In order to observe this property it becomes clear that a meaningful fairness-benchmark-run has to..

1. ..allow every thread i (after execution of $unlock()$) to get back in line for the critical section unhinderedly
2. ..keep track of the order in which the *n* threads have accessed the critical section. We will refer to this data as $lock_{log}$.

By offering a set number of total 'access tickets' to the critical section up for competition, we obtain an unbiased access pattern ($lock_{log}$) compared to when guaranteeing an even split of 'access tickets' to every thread. *(**Example:** 3 threads are in competition. The benchmark allows 9 accesses to the CS. The unbiased access pattern might look like this: **BCBCBCBACC**. Each thread obtains access a differnt number of times. The 'moderated' access pattern on the other handside might enforce an unnatural access pattern, such as **BCBCBCB—AAA**. Because threads B,C have run out of their share of access tickets, A is without competition from '—' forward.)* The statistics obtained from a moderated run are scewed.

1.1.2 Implementation

The sudo code in Figure 1 demonstrates the structure of the fairness-benchmark. The unbiased access pattern of threads is guaranteed by the while pattern in *line 18*, where we let threads enter the competition until $\leq n$ threads are in *line 20* and watining for access (*inner-iterations* is the total number of total 'access tickets'). Within the critical section a thread i is only doing administrative work. Specifically, this includes incrementing the **non-atomic** global index and its local index, but most importantly leaving its mark in the $lock_{log}$.

After at most $inner - iterations$ accesses to the CS the run is over and delivers the observed access order as output.The process is repeated $outer - iterations$ times in order to establish enough data to set up sound statistics regarding overtaking frequencies in a later step.

In order to allow valid argumentation about possible breaching of upper bounds for overtaking the number of global access tickets needs to be large enough. To account for the locks tested $inner - iterations$ has been set to $n * (n + 1)$, where n is the number of competing threads. $outer - iterations$ has been set to 30.

1.1.3 Correctness

A working lock implementation guarantees that at any time only one single thread can execute the critical section. The local access counter $local - counter$ is atomic in the sense that it can only be accessed by one thread (and a single thread can obviously never cause a race condition), whereas the global access counter $global - counter$ is a shared **non-atomic** variable and therefore subject of race conditions. Once the sum of local indices is higher/ unequal the global counter, more than one thread must have been in the critical section at some point. This check is implemented in Figure 1;*lines 37-38*.

1.1.4 Post Processing

We can make a statement about fairness by establishing information about the number of threads *j* obtaining access to the critical section inbetween **two** accesses by thread *i*. In particular, we would expect the number of such accesses to be n in a fair system, in any system we can expoect n to be the median of **waiting-for-lock period lengths** (*gaps*). However, by comparing the distribution of the latter property among different locks and thread counts n (with special respect to the maximum) we compare the situations in fairness. These comparisons will be carried out on the basis of boxplot diagrams in a later section.

In additon we decided to provide a second metric, which certainly has a crosscorrelation to the first (*gaps*), but can provide additional insight into a more basic form of fairness: (*Example: Imagine a lock for which the avg. gap*$< n$*. This implies that one or more threads have* ***not*** *gained access to the cirtical section a single time. Since this* ***very relevant*** *and hugely unfair characteristic of an access pattern might not be observable in the gap statistics, it is useful to add a second more basic fairness metric: The access count of a thread* j *(*$local - access - count$*)compared to the access count thread* j *would have obtained in a fair system (*$\frac{global-access-count}{n}$). With this metric the total distribution of accesses amongst threads can be visualized in a more vivid fashion.

1.2 Throughput

The *Throughput*-metric is a straight-forward performance indication. It provides information about how many accesses to the critical section a lock can achieve per time. In order to set up the benchmarking run in a reasonable way, one has to make sure..

1. that the CS itself holds as little action as possible (otherwise we risk to measure the execution time of the CS rather than the time induced by execution of *(un)lock()*.
2. that we count accesses during full-operation (all threads are constantly re-queueing), as measuremnents might not be scewed if only a few (or a single) thread are competing for the lock at periods.

The experimental setup is only slightly different from the one used for the *Fairness*-metric. While the access pattern is of no concern here, we keep outer- and inner-loop in in place. Before we let n threads compete for somewhere between $n^2 - 1$ and $n^2 - 1 + n$ access tickets, one thread i takes the starting time (*line 18*). The barrier in *line 19* is of crucial importance. In order to respect *(1.)* threads only increase the access counters in the CS (this is necessary for the correctness check). Once all access tickets have been given out, threads meet at a barrier (*line 33*) and the same thread i takes the endtime, computes the throughput and exports it. As with the *Fairness*-examination the experiment is repeated *outer-iteration*-times.

1.3 Latency

The *Latency*-metric is easily missunderstood. While most will agree, that it is a time metric it might be interpreted in several ways. The simplest is to describe it as the inverse of the throughput-metric, hence not answering the question: *how many accesses per time?*, but rather *how much time per access?*. However, this metric would not yield any new insight compared to *Throughput*.
Another approach would be to define latency as the average time it takes a thread from calling *lock()* to being allowed into the critical section. While being closer to the core subject matter, this is inaccurate and indeed wrong, as (for any other than the first thread) the measure taken includes the seconds other threads spend within the critical section.

Finally, we define **latency as the average time it takes the first thread (in the sense of: the first thread to enter the CS) from calling *lock()* to being allowed into the critical section**. This metric holds new insight compared to *Throughput*: (*Example: Locks demanding every thread to run through a fixed number of levels (e.g. filter lock: $\propto n$ levels) might be characterized by linearly increasing (start-up-)latency for higher n (hypothesis!). In comparison, locks that allow fast paths or similar mechanisms might be able to keep this latency constant. This property will not be observably in the Throughput-metric, as the effect cannot be extracted from data generated when the lock is in full operation.*)

Figure 3 illustrates the corresponding Benchmark-construction. Different to the ones presented earlier, we have no need for the threads to re-queue after exiting the critical system. Instead, we attribute a **single access to the critical section to every thread** j and take the measures $latency_j$ it takes the thread to execute the *lock()*-method (*lines 20,23*). Since we only care about the **execution time** $latency_i$ **of the first-entering thread** i(thread i being the thread that encountered *global-count*$= 0$ (*line 24*)), *Latency* $=latency_i$.

The experiment is conducted several *outer-iteration*-times to diminished statistical effects. The check for coorectness is trivial (*line 37*).

```
for (int j = 0; j < outer_iterations; j++)
{
    INIT lock_datastructures;
    INIT lock_log[inner_iterations] = {-1}; // set nonexistent thread-ID
    INIT global_counter = 0;          // global access count (NON ATOMIC!)
    INIT local_counter = 0;           // local access count
    INIT shared_counter[n_threads]; // collection of local access counts

    #pragma omp parallel private(local_counter)
    shared(lock_datastrucures, lock_log, global_counter, shared_counter)
    {
        INIT tid; //thread ID
        RESET local_counter = 0;

        #pragma omp single
        {
        RESET lock_datastructures;
        RESET global_counter=0;
        }
        #pragma omp barrier

        while (global_counter < (n_threads^2 -1)) // MAIN PART
        {
            CALL lock();

            _____CRITICAL SECTION START_____
            lock_log[index] = tid;  //write into log
            global_counter += 1;    //increment global access count
            local_counter += 1;     //increment local access count
            _____CRITICAL SECTION END______

            CALL unlock();
        }

        #pragma omp barrier
        shared_counter[tid] = local_counter; //collect local access count
        #pragma omp barrier

        #pragma omp single
        {
            INIT real_index = SUM(th_indices); //find real acess count
            ASSERT(global_counter == real_index); //correctness check
            OUTPUT log_lock;              //export access order
        }
    }
}
```

Figure 1: Structure of the Fairness-Benchmark

```
for (int j = 0; j < outer_iterations; j++)
{
    INIT lock_datastructures;
    INIT global_counter = 0;          // global access count (NON ATOMIC!)
    INIT local_counter = 0;           // local access count
    INIT shared_counter[n_threads]; // collection of local access counts

    #pragma omp parallel private(local_counter)
    shared(lock_datastrucures, lock_log, global_counter, shared_counter)
    {
        INIT tid;             // thread ID
        INIT time1, time2;
        RESET local_counter = 0;

        #pragma omp single{RESET lock_datastructures, global_counter=0;}

        #pragma omp barrier
        if (tid == 0){ SET time_1} // set starttime right before competition
        #pragma omp barrier

        while (global_counter < (n_threads^2 -1)) // MAIN PART
        {
            CALL lock();

            _____CRITICAL SECTION START_____
            global_counter += 1;     //increment global access count
            local_counter += 1;      //increment local access count
            _____CRITICAL SECTION END______

            CALL unlock();
        }

        #pragma omp barrier // wait until all threads have finished
        if (tid == 0)       // same thread needs to take endtime (local var)
        {
            SET time_2;
            OUTPUT global_counter / (time_2 - time_1); // thtoughput
        }

        shared_counter[tid] = local_counter; //collect local access count
        #pragma omp barrier

        #pragma omp single
        {
            INIT real_index = SUM(th_indices); //find real acess count
            ASSERT(global_counter == real_index); //correctness check
        }
    }
}
```

Figure 2: Structure of the Throughput Benchmark

```
for (int j = 0; j < outer_iterations; j++)
{
    INIT lock_datastructures;
    INIT global_counter = 0;          // global access count (NON ATOMIC!)

    #pragma omp parallel private()
    shared(lock_datastrucures, global_counter)
    {
        INIT tid; //thread ID

        #pragma omp single
        {
        RESET lock_datastructures;
        RESET global_counter=0;
        }
        #pragma omp barrier

        // ERASE INNER LOOP
        SET time_1;
        CALL lock();

        _____CRITICAL SECTION START_____
        SET time_2
        if (global_counter == 0) // 1 for first thread
        {
            OUTPUT (time_2 - time_1) * 1e6; // latency in micros
        }
        global_counter += 1;     //increment global access count
        _____CRITICAL SECTION END______

        CALL unlock();

        #pragma omp barrier

        #pragma omp single // CORRECTNESS CHECK
        {
            ASSERT(global_counter == n_threads); //correctness check
        }
    }
}
```

Figure 3: Structure of the Latency Benchmark

2 Filter Locks

2.1 The role of atomicity

Constrcution of locks assumes sequential consistency. In order to enforce the latter we need to assume atomicity on register level. In implementation this is realized the following way: All (shared) arrays will be of type *atomic-int* from the <*stdatomic.h*> header. (On the otherside *global access count* of course has to remain **non-atomic** - otherwise we cannot prove *Correctness*.

2.2 Standard Filter Lock

The n-thread *Filter*-lock is a construction that demands every thread j to paass through $n-1$ levels until it obtains access to the critical section. Hereby, every level has a certain capacity of threads (level $n-1$ can house only one thread $\rightarrow$ critical section, level 0 holds n threads) and is subject of the same mechanics: As threads try to rise through the ranks, in every level i one of them will be blocked when the rest transitions to level $i+1$. This is facilitated by an **atomic** array (victim), in which a thread j has to enter it's thread-ID at index l once it enters level l. If victim[l] remains j, thread j was the last to enter level l and will be blocked from further transition (*secondary condition*). It only has a chance to proceed once another thread enters level j and takes over the victim-status OR no thread k with $k \neq j$ is located in a higher level j (*primary*[1] *condition*).

2.2.1 Characteristics

The mechanics explained are cause for these characteristics in terms of **Overtaking** and *Fairness*:

1. The number of levels a thread j has to go through is solely determined by n (and not by whether $\exists k$ with level[k]>level[j]). As a result we **expect start-up latency to increase with the number of competing threads** n (there is no fast path in place).

2. As the execution speed of any process is finite but unpredictable (for any parallel execution) **(I)**, the following situation may occur: thread j is victim in level l, then a new thread k calls $unlock()$ and re-enters the race via $lock()$. At this point the *primary condition* would allow j to move on, but the thread remains idle as of (**I**), while k enters level j, takes victim-hood, but still finds the *primary condition* to be unfulfilled and moves in level $j+1$.This effect is called **Overtaking**. Even though j is not victim, infinitely many threads can join thread k and pass j (if k enters i last and leaves first due to *primary condition*). **Overtaking is unbounded**.

2.2.2 Benchmarking Results

Bear in mind that the *Fairness*-benchmarking allows for n^2-1 critical section accesses to be won (this is therefore the technical upper limit for overtaking regardless of the lock's bound). From the *Overtaking*-plot in figure 5 we extract that this rechnical limit is reached exactly for a least one occasions ($n = 13$). The significance of this observation becomes quite conclusive, once compared to the data obtained from the *Block Woo* runs (where **Overtaking** can be witnesses to be bounded). Both locks yield very similar medians for **Overtaking**, but the *Filter*-lock results in way more top-outliers, for which the **Overtaking** count is about an order of magnitude larger compared to the maxima of the *Block Woo* results. **The *Filter lock* is very unfair.**

This impression is supported by the *Accesses*-plot in figure 5. It illustrates how evenly number of accesses to CS are spread aomgst threads (unrespective of order). While the median remains close to the fair number (=expected number) of accesses, some threads have evidently been able

[1] the labels primary and secondary may be inverse in the Alargasamy paper [Ala05]

to obtain an extremely large share of the available accesses. *(Example: For the $n = 32$ run one thread was able to win 30-times what would be the expected access tickets, which is almost all of them (remember: total available stock $n * n$)*[2].

When it comes to *Throughput* the the bottom-plot in figure 5 reveals that the highest scores are obtained for $n < 10$ (the maximum is generated for $n = 2$ and reaches $2.4 * 10^6 \frac{locks}{s}$). These results are in line with what has been witnessed in the *Reference*-lock (**OMP-Lock**), which is able to generate $3 * 10^6 \frac{locks}{s}$ for $n = 4$. For both *Filter*- and *Reference*-lock, performance decreases for higher n and remains $< 0.1 * 10^6 \frac{locks}{s}$ (for the *Filter*-) and $< 0.5 * 10^6 \frac{locks}{s}$ (for the *Reference*-lock).

The most intersting piece of data might be the observed *Latency* (figure 5 - 3). A detailed discussion regarding this can be found in the *Alargasamy Filter*-lock section.

[2] You might be wondering how this plot can hold another outlier which an overtaking count of 26 (as $30 * 32 + 26 * 32 > n^2$), be assured: this is because the plot holds the results of 30 runs.

2.3 Block-Woo Filter Lock

First introduced by Block and Woo [BW90], the inventors not only sticked to the proud developers' habit of creative lock-naming, but were also able to impose a bound on **Overtaking** and reach a better level of fairness compared to the *Standard Filter*-lock.

This lock is constructed a little differently. Rather than forcing thread j though n levels, the thread only has to go through m levels, where m describes the number of threads, which are interested in attaining the CS. Thread j can transition to the next level $i+1$ by the same *secondary condition* presented in the *Filter*-lock (another thread entered level i later) or through a new *primary condition*, which is fulfilled if thread j's level i is high enough to enter the CS (m). After fulfilling requirements for transition to a higher level, but prior to tranisition, the algorithm allows access to the CS if thread j is victim in the old level i (*tertiary condition*). This is to ensure that j is to ascend by power of the *primary condition* (level m is equivalent to level $n-2$ in the *Standard Filter*-lock: If j was to ascend due to *primary condition* only one thread on that level $> m$ can hold victim status, so this satisfies exclusivity). However, if j was to ascend due to non-victimhood (*secondary condition*), thread j cannot possibly satisfy the *tertiary condition*. Concludingly, the combination of *primary* and *tertiary* condition allows a faster entry to the CS, at times when not all n threads are interested.

2.3.1 Characteristics

Deducting from the lock's mechanics described above, *Block-Woo Filter*-lock can be described by a slightly different set of characteristics compared to the *Standard Filter*-lock:

1. We already mentioned that a thread j can enter the CS once it is on a level m that is high enough. We expect this to have great influence on **start-up latency**. Such that it **may stay constant with the number of competing threads** n. Our suspicion is based on the fact, that following (**I**) one thread j might be fastest to declare interest (set *Q[j]*) at the start of the competition AND check for the *secondary condition* fast enough (before any other thread can set *Q[i]*) after. It would then read *interested threads=1* with it's level $i=1$ and would immediately get acces to the CS.

2. With respect to **Overtaking** dynamics are slightly different than for the *Standard Filter*-lock. In the latter's *lock()* thread j could be overtaken an unbounded number of times, because the *primary condition* allows infinitely many threads to pass by if only the overtaking thread k, that enters last, also ascends first. This dynamic is no longer possible as the new *primary condition* (number of interested threads $\leq$ level) cleary forces the same last thread k that enters level i to remain there (whereas the thread j is free to ascend). In the worst case this new upwards-cascade continoues until thrad j reaches level $m=n$ and has been overtaken n times by $n-1, n-2, .., 1$ threads. **This results in an upper bound of $\frac{n*(n-1)}{2}$ for Overtaking**.

2.3.2 Benchmarking Results

As mentioned in the corresponding section of the *Standard Filter*-lock the *Block Woo Filter*-lock constructional advantages bear fruits in terms of *Fairness*. As put forward in the *Overtaking* graph of figure 6, **Overtaking** does indeed never exceed the theoretical bound of $\frac{n*(n-1)}{2}$. Infact, it **never even comes close to the upper bound** *(Example $n=32$: The upper bound would be 500, the largest outlayer is 40).* This comes as a surprise. While statistically it is expected to not witness a thread that experienced **Overtaking** of a magnitude that exactly fits the upper bound, but not even observing a single outlayer in 12x30 runs in the expected order of magnitude is odd.

Are the odds that little for such an event to happen?

Be that as it may, the *Accesses* plot in figure 6 fittingly illustrates, how incredibly fair the experimental data showed the lock to be. Despite some outlayers for $n = 16$ the lock seems to induce a relatively even split of access tickets among threads.

With respect to *Throughput* the the bottom-plot in figure 6 indicates the cost of *Fairness*. While the performance behaviour along n is comparable to *Standard Filter*- and the *Reference*-lock, absolute performance is lower. A maximum at $n = 3$ with *Throughput*$= 1.7 * 10^6 \frac{locks}{s}$. Performance decreases for higher n and runs at only $< 0.1 * 10^6 \frac{locks}{s}$ (but even below levels of the *Standard Filter*-lock).

2.4 Alargasamy Filter Lock

First introduced by Alargasamy in the [Ala05] paper, this lock is an improved version of the *Filter*-locks presented so far. The creators introduce the idea of 'promoting' to the *Filter*-lock, this is: When the thread j (who got into the CS from level g) leaves the critical section/ calls *unlock()* it will set *victim[l]=j*[3] for all levels $l < g$. The mechanism allows ALL threads sitting in the levels l to ascend and get closer to level m. **Overtaking** becomes less like, as now it can only occur 'relative to natural ascending speed of the mass of threads' (visually speaking).
However, as now multiple threads could theoretically get into level m (at the same time), the creators needed to introduce a protection mechanism againt this: The *Standard Filter*-lock's *primary condition* is replaced by the combined *primary conditions* from *Standard and Block Woo Filter*-locks, hence:
'thread j can enter ascend if it reaches a level m, high enough to enter the CS AND all other threads are in lower levels'
Same as in the *Block Woo Filter*, the *tertiary condition* prohibits *secondary condition*-powered access to the CS (the latter is only here for ascention through the levels).

2.4.1 Characteristics

The dynamics by which the *Alargasamy Filter*-lock is governed have changed drastically compared to it's predecessors (*Block-Woo/Standard Filter*-lock):

1. Due to the promoting mechanism introduced above, threads do no have to rely solely on ascention powered by *secondary condition* (another thread assuming victimhood). A thread j blocked in the first stage, will consequently be promoted by all $n-1$ other threads and end up in level $m = n$, where it can get access. As this is the worst case, the upper bound for **Overtaking** should be $n-1$ - way lower than the *Block Woo*-bound and indeed optimal.

2. We expect *Latency* to be similar to the *Block Woo*-lock, since the start-up process is no different.

2.4.2 Benchmarking and Results

The **Overtaking**-plot in 7 confirms the author's claim as to delivering the optimal bound for the unfair effect: $n-1$ does indeed hold for all n. The *Accesses*-plot speaks to the fair distribution of access tickets in absolute terms.

The performance decay in terms of *Throughput* sets in earlier than it does for the *Block-Woo*-lock (n 8), the stationary performance for larger n, however, is not significantly worse, when compared to *Block-Woo*. The *Alargasamy*-lock might very well be a sensible replacement with exceptional fairness-characteristics.

Latency

Finally, *Latency* results for *Block Woo Filter*- (figure 6 - 3) and *Standard Filter*-lock (figure 5 - 3) bring up questions. For both locks, the time it takes the first-winning thread to get hold of the CS grows almost exponentially for $0 < n < 10$ and grows at a lower rate after. The same holds for the *Alargasamy Filter*-lock (figure 7 - 3). While our thesis for the *Standard Filter*-lock(latency increases with n) holds, our thesis for the *Block Woo Filter*-lock (latency might remain constant) does not. Apparently. the first thread to set *Q[j]=1* does not get to inquire the *primary condition* prior to other threads announcing their interest in accessing the CS (*Q[i]=1*), too - Why ?

[3]The authors made an indexing error in *line 8* of their sudo-code

In any case, it is surprising to see that the cost to rise through the levels increases this fast. We stress this might be due to the *Atomicity* of the *victim,level* registers. For small n the number of threads bounds *Latency*, whereas for larger n the speed of information retrieval from the registers (through constant issueing of *read()* by waiting threads) determines the bound. However, if that was the case, we would expect all three locks to hit the same ceiling after a while - but while *Alargasamy Filter*-lock and *Block Woo Filter*-lock do show very similar behaviour, the *Standard Filter*-lock's *Latency*, remains smaller by one order of magnitude.

Additionally, when comparing to the locks without 'level construction', we have to point out that both the *Competition Tree*-lock and the *Reference*-lock show linear latency-growth. We remain unaware of the reason for the lofty growth of *Latency* for the 'level featuring' locks.

3 Binary tree - assembled 2-thread-Peterson-locks

In the leaf locks each thread i has to compete with one other thread j. The winner moves into the next level of locks and continues all the way to the root-lock (acquire() method), from where it enters the CS. Every sub-lock lock along the way is a 2-thread Peterson lock and treats one thread as $i = 0$ and the other as $j = 1$ 1. Upon exit from the CS, release() unlocks each of the 2-thread Peterson locks the thread has acquired (from the leaf to root).

3.1 Implementation

Since our implemetation uses pure C, an object-oriented approach is not possible and we are restricted to using a more functional approach: Instead of a shared boolean `flag[2]`-array and the shared variable `victim` (as in use in the 2-thread Peterson lock), we expand `flag_tree[`$2 \cdot (n-1)$`]` and replace `victim` by a `victim_tree[`$n-1$`]`-array.

n is the number of threads and `n-1` is the number of locks in the tree (assuming n is a power of 2). The depth of the binary tree is `levels = ceil(log2(n))`, where we again assume n to be a power of two or rounded up otherwise. For the latter case (arbitrary n), the number of locks is $2^{\texttt{ceil(log2(n))} - 1}$ (the closest power of $2 > n$).

If thread j wins a sub-lcok in level l and gets into $l + 1$ it gets assigned a sub-lock of level $l + 1$ (a certain partition in`flag_tree` and `victim_tree`) based on it's threadID and $l + 1$). The partitions starting at index `victim_offset` (for `victim_tree`) and index `flag_offset` (for `flag_tree`) are computed as follows:

$$\texttt{victim\_offset} = \frac{\texttt{floor}(i/2) + (2^{level} - 1) * (2^{levels})}{2^{level}}$$

$$\texttt{flag\_offset} = 2 * \texttt{victim\_offset}$$

$$i = \frac{\texttt{floor(threadId)}}{2^{level}}$$

where $level$ is the current out of a total of $levels$ levels and i is a "level-local" thread ID.
Essentially, a thread j calls the standard 2-thread Peterson lock with `&flag[flag_offset]` `&victim[victim_offset]` and `i%2`[4] in each level.

[4] map to either 0 or 1

3.2 Implementation Case Study

The idea shall be illustrated for the 8-thread case, but generailzes to an arbitrary number of threads. Figure 4 shows the ”flow” of thread 5 through the tree, up to the root lock:

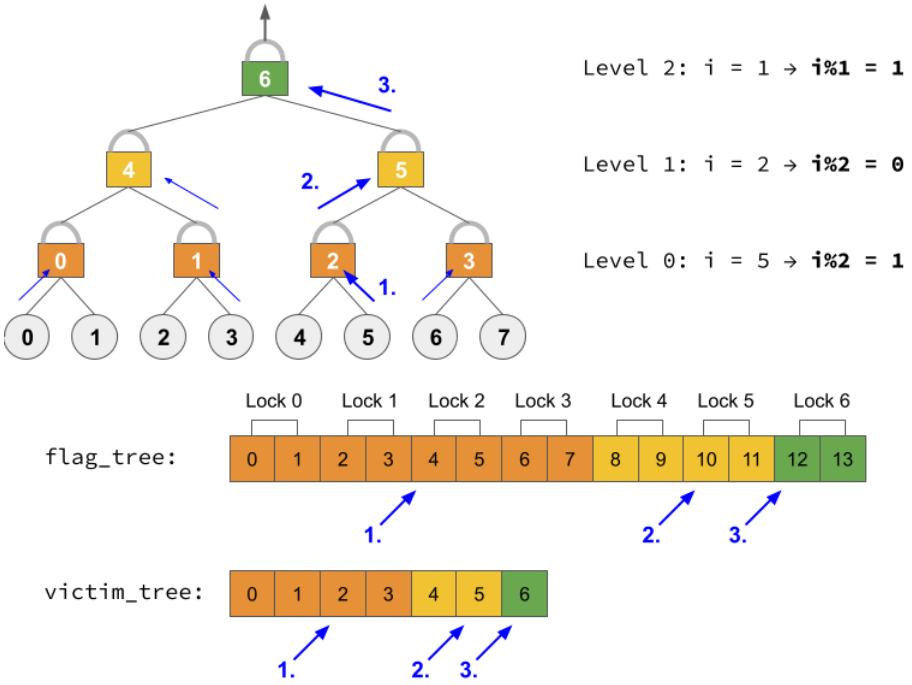

Figure 4: Tree lock: Illustrative case study.

Orange ≡ level 0, **Yellow** ≡ level 1 and **Green** ≡ level 3.

The upper graphic in Figure 4 shows the sub-lock-embedding in the tree, where the color encodes the level the sub-lock is part of.

The lower graphic shows the `flag_tree` and `victim_tree` arrays. The **blue** arrows in both graphics denote in which level (≡ loop iteration) - thread 5 is:
Based on it's thread ID and the current level 0, thread 5 computes `victim_offset`= 2, `flag_offset`= 4 (which correspond to lock 2). Thread 5's level-local thread ID $i = 5$, which gets maped to $i\%2 = 1$ (right side entry to lock 2).In level 1, thread 5 computes the following offsets: `victim_offset`= 5, `flag_offset`= 10 (corresponding to lock number 5). Further: $i = 2$ and $i\%2 = 0$ (leftside entry to lock 5). Finally in the last level(level 2) we obtain: `victim_offset`= 6, `flag_offset`= 12 (corresponding to lock 6) for thread 5. Following $i = 1$ with $i\%2 = 1$ (right side entry to lock 6).

3.3 Characteristics

The tournament tree lock provides **no bounds on Overtaking**. Threads acquiring the lock can immediatly re-apply since the locks are released from leaf to root lock. This could, in theory, lead to situations where one thread acquires the lock infinitely often.
If we assume a somewhat fair **sub-**lock acquisition order, we could argue that the expected *Latency* for a the first thread to acquire the lock should be $\propto$ *ceil(log2(n))* (number of levels in the tree). A similar argument can be made for throughput. Again, this only holds under the assumption of fairnees and that all threads on the same level acquire their respective locks concurrently.

3.4 Benchmarking Results

The lack of bounds for **Overtaking** becomes apparent in Figure 8: The first fairness plot shows the greatest outliers among all locks, **exceeding even the Filter lock**. The median is in line with the other locks (some threads seem to acquire the lock multiple times in a row, while others are completely left out- basic statistic). The impression is supported by the second picture: In absolute terms this is the unfairest lock, except for the *Reference*-lock.

Within the range of the standard deviation, the theoretic behavior for the *Latency* (*ceil(log2(n))*) with growing number of threads seems to confirm the assumptions made. The *Throughput* of the tree lock decays roughly exponentially with the number of competing threads, which is also in line with assumption of logarithmic behaviour (inverse that is, thus the exponential decay). The performance is comparable to the *Standard Filter*-lock. It seems the 'unfair' locks perform slightly better overall - *Fairness* is costly.

3.5 Notes on Correctness

While testing and benchmarking our tree lock implementation, we realised that in some cases, the lock is **not able** to achieve 100% mutual exclusion. It is hard to determine the exact cause, as all relevant registers are **atomic** and the 2-thread Peterson lock on it's own runs **flawlessly**.As a result, we relaxed the conditions for this lock and **removed the assert statement in the Benchmark** in order to obtain results. Still, we want to provide some heuristics of why we think the tree lock works in principle:

We do so by letting **10 threads** compete for the lock inner-iterations =**100.000** -times and outer-iterations= 5 (this resembles the *Throughput*-benchmark explained above). A Correctness-Check as explained in a prior section follows:

Lock	shared counter	$\sum$ local counters	expected value
Tree lock:	993930	1000000	1000000
	999435	1000000	1000000
	999558	1000000	1000000
	997364	1000000	1000000
	997447	1000000	1000000
No lock:	122302	1000000	1000000
	113410	1000000	1000000
	132477	1000000	1000000
	118612	1000000	1000000
	127437	1000000	1000000

The difference between using the tree lock vs. no lock at all is **clearly observable**. In the case of no lock, we have almost perfect race conditions, with the shared counter being only slightly above 100.000. When using the tree lock, the shared counter comes very close to the expected value of 10 times the prescribed locks, i.e. 1.000.000. It remains unclear where this disturbance stems from (we suspect some sort of compiler raitonalisation we are unable to observe).

A Appendix

B Source Code

1. To run *small-bench* and *small-plot*, please follow the instructions in the *README*
2. take special notice of the information regarding *gcc-* compiler version

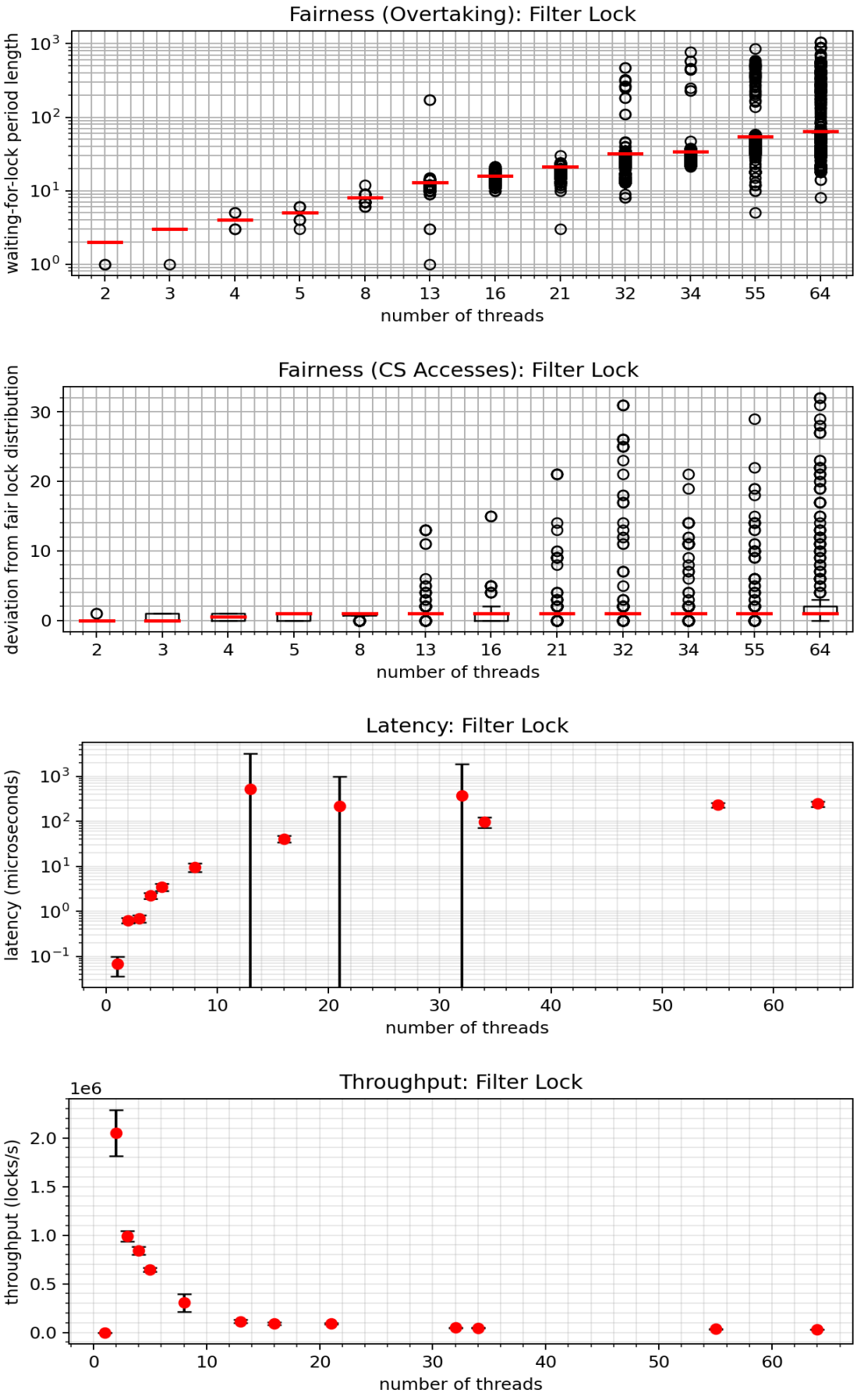

Figure 5: Benchmarking Results: Filter Lock

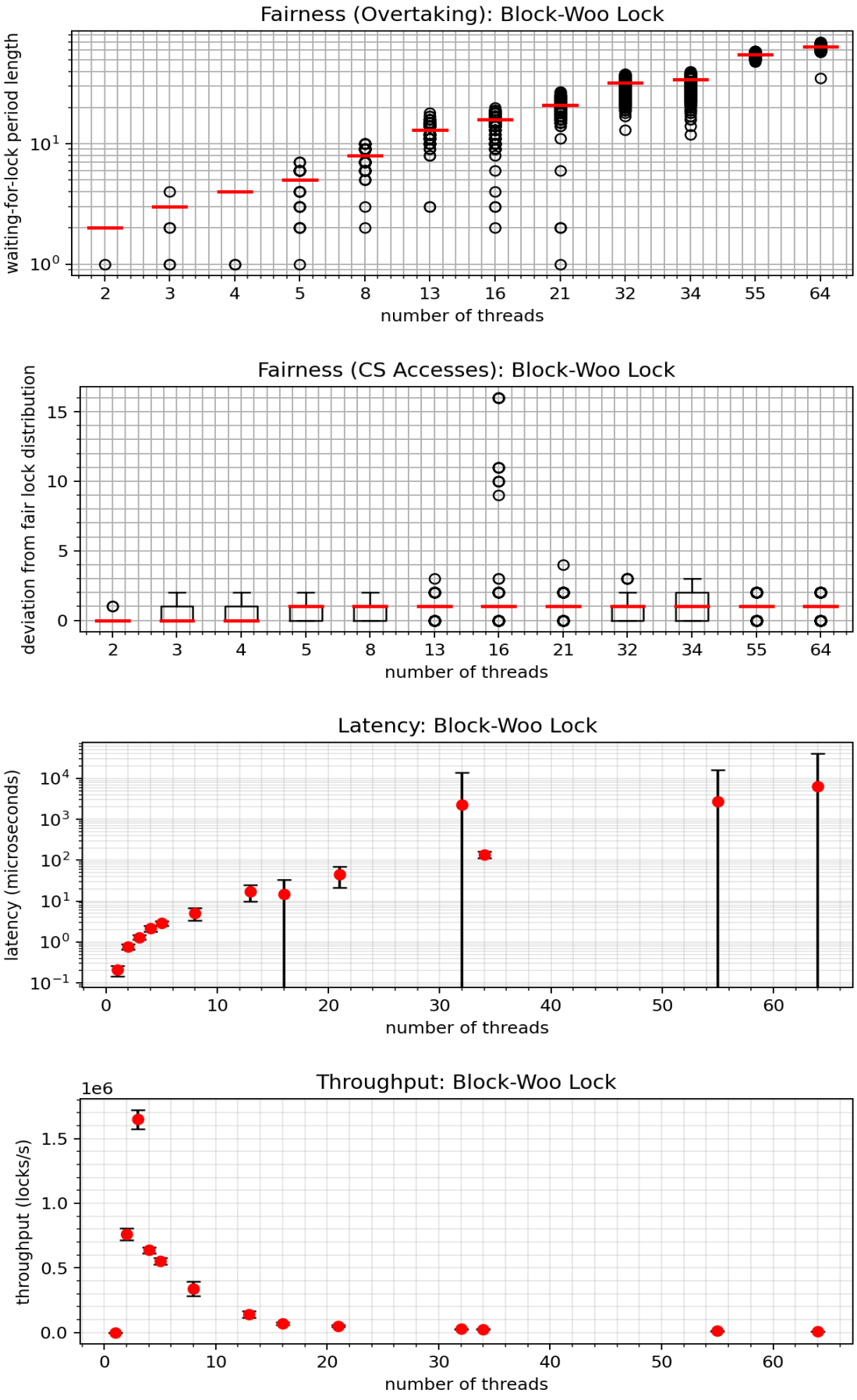

Figure 6: Benchmarking Results: Block Woo Lock

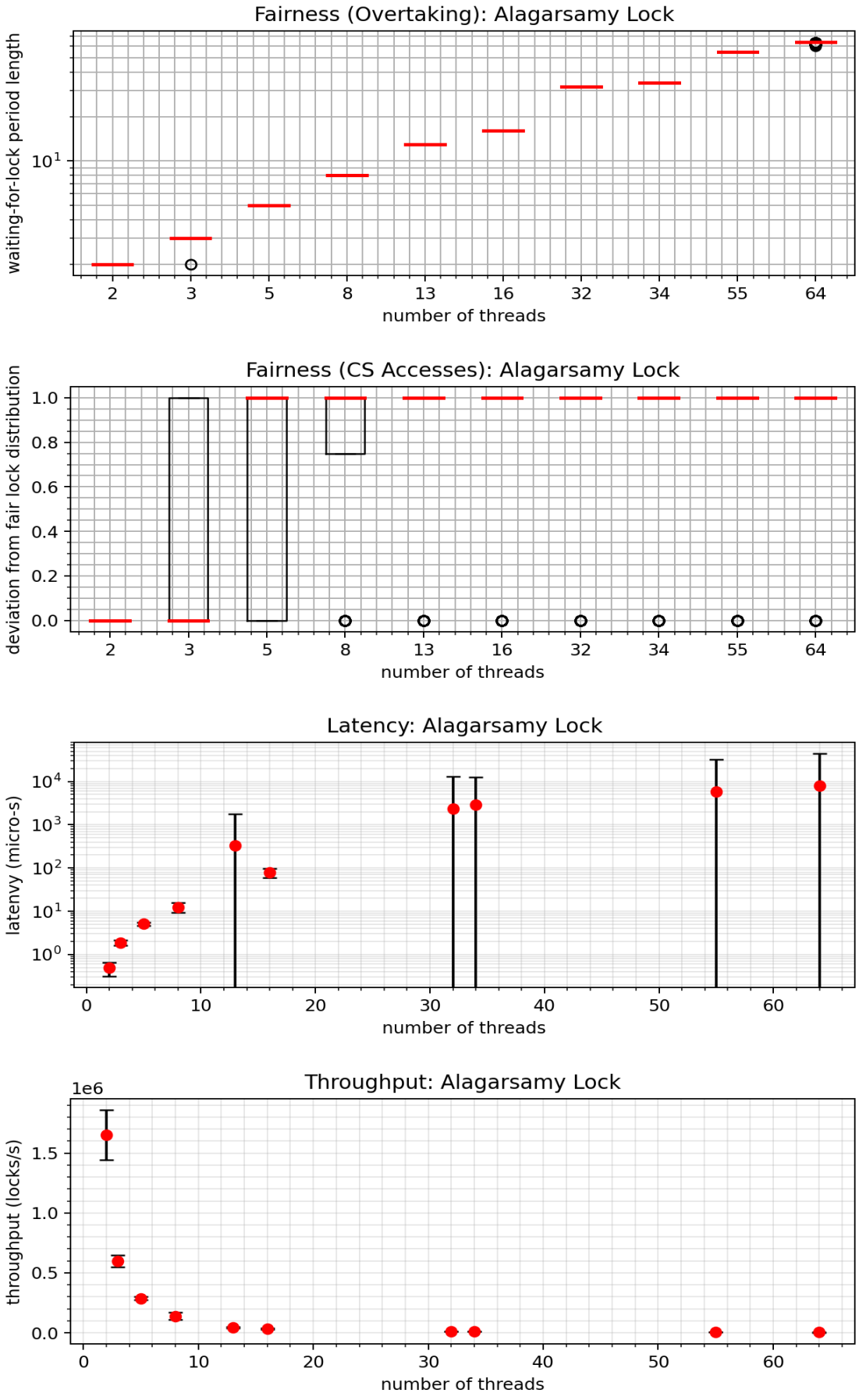

Figure 7: Benchmarking Results: Reference Alargasamy Lock

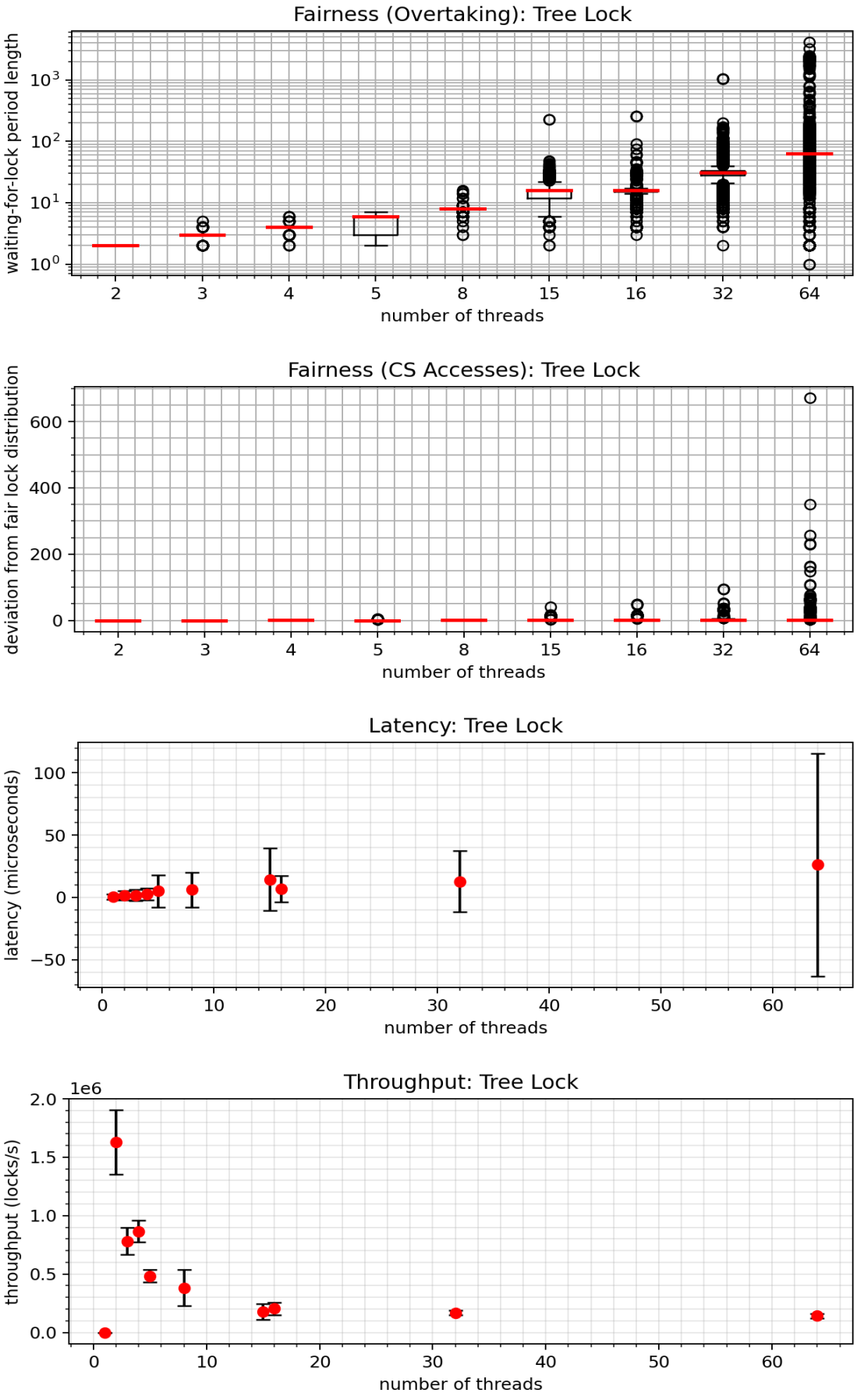

Figure 8: Benchmarking Results: Binary Tree Lock

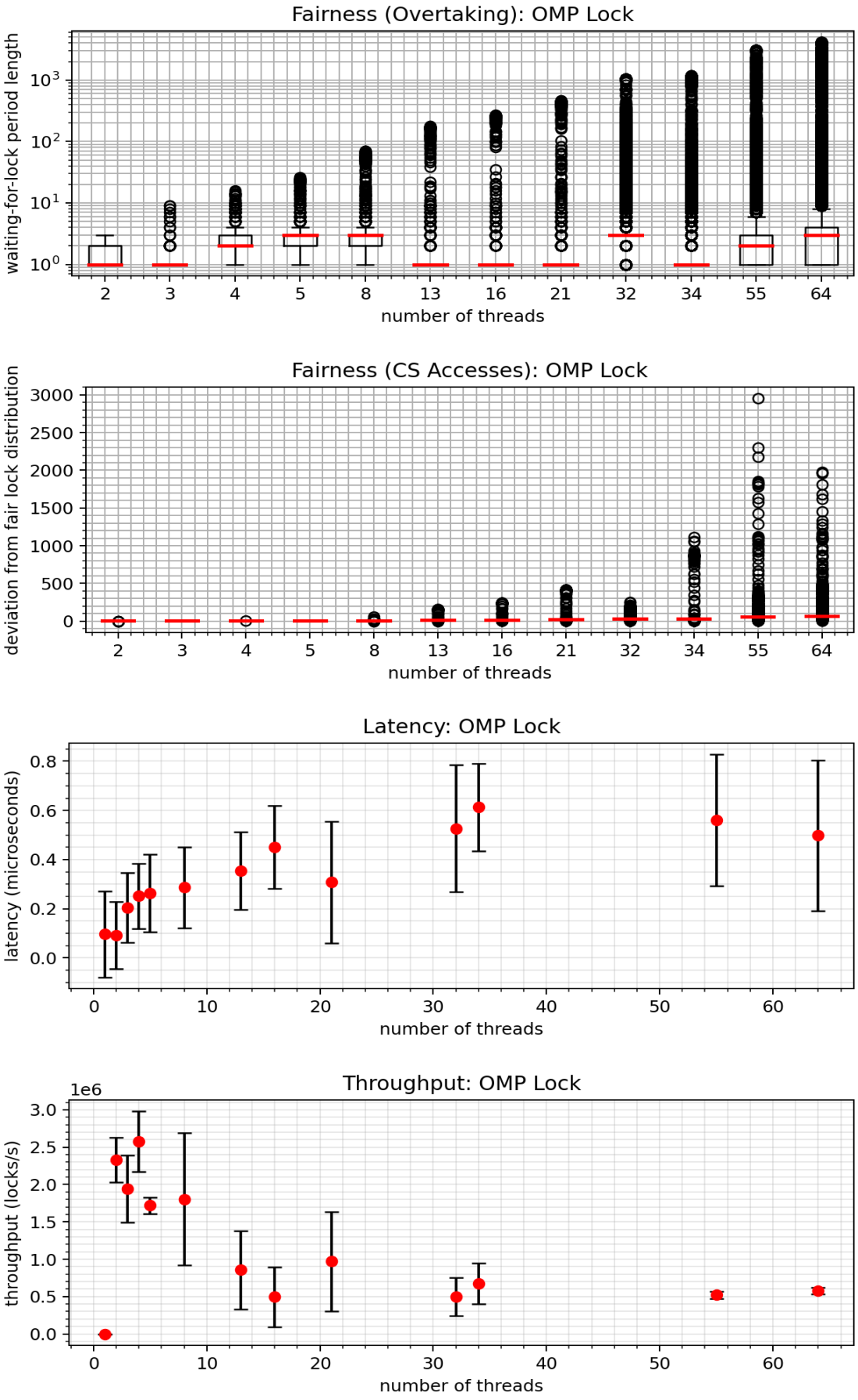

Figure 9: Benchmarking Results: Reference OMP Lock