# Brief Announcement: Performance Prediction for Coarse-Grained Locking

Vitalii Aksenov, Dan Alistarh, Petr Kuznetsov

# Brief Announcement: Performance Prediction for Coarse-Grained Locking

Vitaly Aksenov
ITMO University and Inria Paris

Dan Alistarh
IST Austria

Petr Kuznetsov
LCTI, Télécom ParisTech, Université Paris-Saclay

## ABSTRACT

A standard design pattern found in many concurrent data structures, such as hash tables or ordered containers, is an alternation of parallelizable sections that incur no data conflicts and critical sections that must run sequentially and are protected with locks. A lock can be viewed as a *queue* that arbitrates the order in which the critical sections are executed, and a natural question is whether we can use *stochastic analysis* to predict the resulting throughput. As a preliminary evidence to the affirmative, we describe a simple model that can be used to predict the throughput of *coarse-grained* lock-based algorithms. We show that our model works well for CLH lock, and we expect it to work for other popular lock designs such as TTAS, MCS, etc.

**ACM Reference Format:**
Vitaly Aksenov, Dan Alistarh, and Petr Kuznetsov. 2018. Brief Announcement: Performance Prediction for Coarse-Grained Locking . In *PODC '18: ACM Symposium on Principles of Distributed Computing, July 23–27, 2018, Egham, United Kingdom.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3212734.3212785

## 1 ABSTRACT COARSE-GRAINED SYNCHRONIZATION

Conventionally, the performance of a concurrent data structure is evaluated via experiments, and it is notoriously difficult to account for all significant experimental parameters so that the outcomes are meaningful. Our motivation here is to complement experimental evaluation with an analytical model that can be used to *predict* the performance rather than measure it. As a first step towards this goal, in this work, we attempt to predict the throughput of a class of algorithms that use *coarse-grained* synchronization.

Consider a concurrent system with $N$ processes that obey the following simple *uniform* scheduler: at every time step, each process performs a step of computation. This scheduler, resembling the well-known PRAM model [3], appears to be a reasonable approximation of a real-life concurrent system. Suppose that the processes share a data structure exporting a single `operation()`. If the operation induces a work of size $P$ and incurs no synchronization, the resulting throughput is $N \cdot \alpha/P$ operations in a unit of time: each process performs $\alpha/P$ operations in a unit of time, where $\alpha$ indicates the amount of work that can be performed by one process


*PODC '18, July 23–27, 2018, Egham, United Kingdom*

ACM ISBN 978-1-4503-5795-1/18/07.
https://doi.org/10.1145/3212734.3212785

```
1 operation():
2   lock.lock()
3   for i in 1..C:
4     nop
5   lock.unlock()
6   for i in 1..P:
7     nop
```

**Figure 1: The coarse-grained operation**

in a unit of time. One way to evaluate the constant $\alpha$ experimentally is to count the total number $F$ of operations, each of work $P$, completed by $N$ processes in time $T$. Then we get $\alpha = F/NP$. The longer is $T$, the more accurate is the estimation of $\alpha$.

Now suppose that, additionally, the operation performed by each process contains a *critical section* of size $C$. In the operation, described in Figure 1, every process takes a global lock, performs the critical section of size $C$, releases the lock and, finally, performs the *parallel section* of size $P$.

Here, as a unit of work, we take the number of CPU cycles spent during one iteration of the loop in Lines 3-4 or 6-7. The iteration consists of a `nop` instruction, an increment of a local variable and a conditional jump, giving us, approximately, *four* CPU cycles in total.

## 2 MODEL ASSUMPTIONS

Below we list basic assumptions on the abstract machine used for our analytical throughput prediction.

First, we assume that coherence of caches is maintained by a variant of MESI protocol [5]. Each cache line can be in one of four states: `Modified` (`M`), `Exclusive` (`E`), `Shared` (`S`) and `Invalid` (`I`). MESI regulates transitions between states of a cache line and responses depending on the request (read or write) to the cache line by a process or on the request to the memory bus. The important transitions for us are: (1) upon reading, the state of the cache line changes from any state to `S`, and, if the state was `I`, then a *read request* is sent to the bus; (2) upon writing, the state of the cache line becomes `M`, and, if the state was `S` or `I`, an *invalidation request* is sent to the bus.

We assume that the caches are *symmetric*: for each MESI state $st$, there exist two constants $R_{st}$ and $W_{st}$ such that any read from any cache line with status $st$ takes $R_{st}$ work and any write to a cache line with status $st$ takes $W_{st}$ work. David et al. [2] showed that for an Intel Xeon machine (similar to the one we use in our experimental validation below), given the relative location of a cache line with respect to the process (whether they are located on the same socket or not), the following hypotheses hold: (1) writes induce the same work, regardless of the state of the cache line; (2) `swaps`, not concurrent with other `swaps`, induce the same work as writes. Therefore, we assume that (1) $W = W_M = W_E = W_S = W_I$ and (2) any contention-free `swap` induces a work of size $W$.

```
1  class Node:
2    bool locked
3
4  Node head = new Node()  // global
5  Node my_node            // per process
6     my_node.locked ← true
7
8  operation():
9    Node next ← swap(&head, my_node)  // W or X
10   while (next.locked) {}            // R_I or 2 · R_I
11   for i in 1..C:                    // C
12     nop
13   my_node.locked ← false            // W
14   my_node ← next
15   my_node.locked ← true             // W
16   for i in 1..P:                    // P
17     nop
```

**Figure 2: The coarse-grained operation with inlined `lock` and `unlock` functions**

# 3 CLH LOCK

Multiple lock implementations have been previously proposed, from simple spinlocks and TTAS to more advanced MCS [4] and CLH [1]. For our analysis, we choose CLH, as the simplest lock among those considered to be efficient. In Figure 2, we inline `lock` and `unlock` calls to CLH lock in our abstract coarse-grained operation.

## 3.1 Cost of an operation

Let us zoom into what happens during the execution of the operation.

Note that at the beginning of an operation (unless it is the very first invocation), `my_node.locked` is loaded into the cache and the corresponding cache line is in state `M`, because of the set in Line 15 during the previous operation by the same process.

(1) The operation starts with `swap` (Line 9) that induces a work of size $W$, if not concurrent with other `swaps`, and a work of size at most $X$, otherwise.

(2) In Line 10, the algorithm loops on a field `next.locked`. During this loop one or two cache misses happens.

One cache miss can happen at the first iteration of the loop if the read of `locked` returns `true`. The last process that grabbed the lock already invalidated this cache line in Line 15 during its penultimate operation. MESI reloads the cache line and changes its state from `I` (or none if it was not loaded previously) to `S`.

The other cache miss happens in every execution when the operation reads `next.locked` and gets `false`. In this case, the cache line was invalidated in Line 13 during the last operation of the last process that grabbed the lock. MESI reloads the cache line and changes its state from `I` (or none) to `S`.

Each of the described cache misses induces the work of size $R_I$. Thus, the work induced in Line 10 is of size of $R_I$ (if only the second miss happens) or $2 \cdot R_I$ (if both misses happen).

(3) In Lines 11-12, the critical section with work of size $C$ is performed.

(4) In Line 13, `my_node.locked` is set to `false`. There are two cases: if `my_node.locked` is not yet loaded by any other process in Line 10 then the state remains `M`; otherwise, MESI changes the state from `S` to `M` and sends a signal to invalidate this cache line. In both cases, the induced work is of size $W$.

(5) In Line 14, the operation performs an assignment on local variables, without contributing to the total work.

(6) In Line 15, `my_node.locked` is set to `true`. From the end of the while loop at Line 10 the corresponding cache line is in state `S`. MESI changes the state to `M` and sends a signal to invalidate this cache line inducing work of size $W$.

(7) In Lines 16-17, the parallel work of size $P$ is performed.

## 3.2 Evaluating throughput

To evaluate the throughput of the resulting program under the uniform scheduler, take a closer look on how $N$ processes continuously perform the operation from Figure 2.

Process 1 executes: its first swap (taking at most $X$ units); the critical section (blue, Lines 10-13): acknowledges the ownership of the lock by reading `false` in Line 10 (takes $R_I$ units), performs the work of size $C$ and releases the lock in Line 13 (takes $W$ units); the parallel section (red, Lines 15-17 and 9): sets `my_node.locked` to `true` (takes $W$), performs the work of size $P$, performs a non-contended swap (takes $W$) and, possibly, reads `true` in Line 10 (takes $R_I$). (Here, the swap operation performed after the very first completed critical section is counted in the parallel work, as it is executed in the absence of contention.) Every other process $i$ operates in the same way: it swaps as early as possible (taking at most $X$), waits until process $i-1$ releases the lock, and then performs its critical (blue) and parallel (red) sections.

Depending on the parameters $N$, $C$, $P$, $W$, and $R_I$, two types of executions are possible.

In case 1 (Figure 3a), at the moment when process 1 finishes its parallel section, process $N$ already finished its critical section, i.e., $P + 2 \cdot W > (N-1) \cdot (C + R_I + W)$. Therefore, in the steady case, at every moment of time, each process do not wait and execute either the parallel or critical section, and the read in Line 10 cannot return `true` because the lock is already released. Thus, the throughput, measured as the number of operations completed in a unit of time, equals to $N \cdot \frac{\alpha}{(P+2\cdot W)+(C+R_I+W)}$.

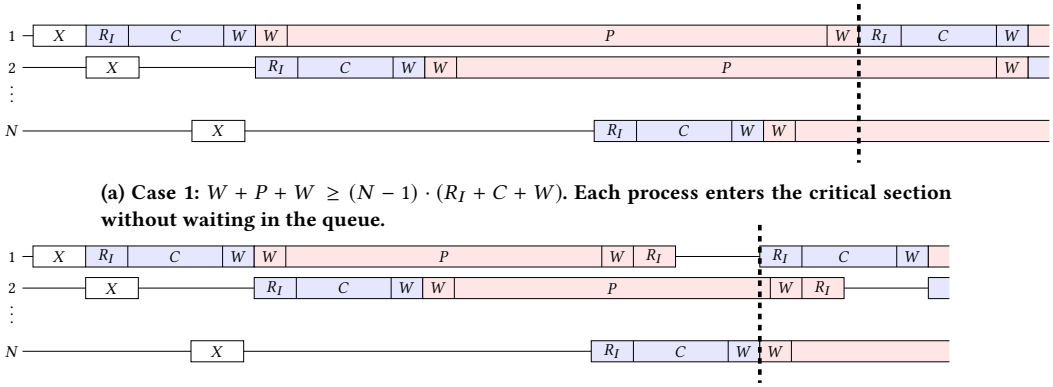
In case 2 (Figure 3b), before proceeding to the next operation, process 1 has to wait until process $N$ completes its critical section from the previous round of operations; process 2 waits for process 1, process 3 waits for process 2, etc. Thus, there is always some process in the critical section, giving the throughput of $\frac{\alpha}{C+R_I+W}$.

Therefore, given the number of processes $N$, the sizes $C$ and $P$ of critical and parallel sections, the throughput can be calculated as follows:

$$\begin{cases} \frac{\alpha}{C+R_I+W} & \text{if } P + 2 \cdot W \le (N-1) \cdot (C + R_I + W) \\ \frac{\alpha \cdot N}{(P+2\cdot W)+(C+R_I+W))} & \text{otherwise} \end{cases}$$

# 4 EXPERIMENTS

For our measurements, we used a server with four 10-core Intel Xeon E7-4870 chips of 2.4 GHz (yielding 40 hardware processes in total), running Ubuntu Linux kernel v3.13.0-66-generic. We compiled the code with MinGW GCC 5.2.0 (with -O0 flag to avoid compiler optimizations, such as function inlining, that can screw up our benchmarking environment). The code is available at https://github.com/Aksenov239/complexity-lock-with-libslock.




**(a) Case 1: $W + P + W \geq (N-1) \cdot (R_I + C + W)$. Each process enters the critical section without waiting in the queue.**




**(b) Case 2: $W + P + W \leq (N-1) \cdot (R_I + C + W)$. Each process waits in the queue before entering the critical section.**

**Figure 3: Examples of executions of the coarse-grained algorithm from Figure 2. Blue intervals depict critical sections and red intervals depict parallel sections.**

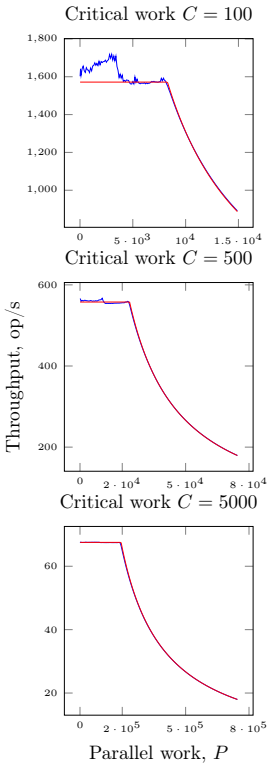



**Figure 4: Throughput on 39 processes for $C \in \{100, 500, 5000\}$**

We considered the following experimental settings: the number of processes $N \in \{5, 10, 20, 30, 39\}$; the size of the critical section $C \in \{100, 500, 1000, 5000, 10000\}$; and the multiplier $x \in [1, 150]$ (we choose all integer values) that determined the size of the parallel section $P = x \cdot C$. For each setting, we measured the throughput for 10 seconds. Our experimental evaluation gives $\alpha \approx 3.5 \cdot 10^5$, $W \approx 40$, and $R_I \approx 80$. The ratio between $W$ and $R_I$ correlates with the experimental results provided by David et al. [2].

In Figure 4 we show our experimental results for three settings with $N = 39$ and $C \in \{100, 500, 5000\}$ (blue curves) compared with our theoretical prediction (red curves). The two curves match very closely, except for the case of small $C$ and $P$ where our predicted throughput underestimates the real one. We relate this to the fact that we oversimplified the abstract machine: any write induces the work of constant size $W$, regardless of the relative location of the cache line with respect to the process. For small $C$ and $P$ two processes from the same socket are more likely to take the lock one after the other and, thus, on average, a write might induce less work than $W$, and, consequently, the throughput can be higher than predicted.

## 5 CONCLUSION

In this short note, we showed that a simple theoretical analysis may quite accurately predict the throughput of data structures implemented using coarse-grained synchronization. For the moment, our analysis is restricted to algorithms using CLH-based locking in systems obeying the uniform scheduler. In upcoming work, we intend extend the analysis to more realistic algorithm designs, lock implementations and architectures.

## 6 ACKNOWLEDGEMENTS

This research is partially supported by European Research Council (ERC-2012-StG-308246) and the Franco-German DFG-ANR Project DISCMAT (14-CE35-0010-02).

## REFERENCES

[1] Travis Craig. 1993. *Building FIFO and priorityqueuing spin locks from atomic swap.* Technical Report. ftp://ftp.cs.washington.edu/tr/1993/02/UW-CSE-93-02-02.pdf
[2] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 33–48.
[3] Joseph JáJá. 1992. *An introduction to parallel algorithms*. Vol. 17. Addison-Wesley Reading.
[4] John M Mellor-Crummey and Michael L Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)* 9, 1 (1991), 21–65.
[5] Mark S Papamarcos and Janak H Patel. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. In *ACM SIGARCH Computer Architecture News*, Vol. 12. ACM, 348–354.