Analyzing the Performance of Lock-Free Data Structures: A Conflict-based Model

Aras Atalar, Paul Renaud-Goud, Philippas Tsigas

Analyzing the Performance of Lock-Free Data Structures: A Conflict-based Model

Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas

Chalmers University of Technology
{aaras|goud|tsigas}@chalmers.se

Abstract. This paper considers the modeling and the analysis of the performance of lock-free concurrent data structures that can be represented as linear combinations of fixed size retry loops.
Our main contribution is a new way of modeling and analyzing a general class of lock-free algorithms, achieving predictions of throughput that are close to what we observe in practice. We emphasize two kinds of conflicts that shape the performance: (i) hardware conflicts, due to concurrent calls to atomic primitives; (ii) logical conflicts, caused by concurrent operations on the shared data structure.
We propose also a common framework that enables a fair comparison between lock-free implementations by covering the whole contention domain, and comes with a method for calculating a good back-off strategy.
Our experimental results, based on a set of widely used concurrent data structures and on abstract lock-free designs, show that our analysis follows closely the actual code behavior.[1]

1 Introduction

Lock-free programming provides highly concurrent access to data and has been increasing its footprint in industrial settings. Providing a modeling and an analysis framework capable of describing the practical performance of lock-free algorithms is an essential, missing resource necessary to the parallel programming and algorithmic research communities in their effort to build on previous intellectual efforts. The definition of lock-freedom mainly guarantees that at least one concurrent operation on the data structure finishes in a finite number of its own steps, regardless of the state of the operations. On the individual operation level, lock-freedom cannot guarantee that an operation will not starve.

The goal of this paper is to provide a way to model and analyze the practically observed performance of lock-free data structures. In the literature, the common performance measure of a lock-free data structure is the throughput, *i.e.* the number of successful operations per unit of time. It is obtained while threads are accessing the data structure according to an access pattern that interleaves local

[1] The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2013-2016) under grant agreement 611183 (EXCESS Project, www.excess-project.eu).

work between calls to consecutive operations on the data structure. Although this access pattern to the data structure is significant, there is no consensus in the literature on what access to be used when comparing two data structures. So, the amount of local work (that we will refer as parallel work for the rest of the paper) could be constant ([14,15]), uniformly distributed ([10], [7]), exponentially distributed ([17], [8]), null ([12,13]), *etc.* More questionably, the average amount is rarely scanned, which leads to a partial covering of the contention domain.

We propose here a common framework enabling a fair comparison between lock-free data structures, while exhibiting the main phenomena that drive performance, and particularly the contention, which leads to different kinds of conflicts. As this is the first step in this direction, we want to deeply analyze the core of the problem, without impacting factors being diluted within a probabilistic smoothing. Therefore, we choose a constant local work, hence constant access rate to the data structures. In addition to the prediction of the data structure performance, our model provides a good back-off strategy, that achieves the peak performance of a lock-free algorithm.

Two kinds of conflict appear during the execution of a lock-free algorithm, leading to additional work. Hardware conflicts occur when concurrent operations call atomic primitives on the same memory location: these calls collide and conduct to stall time, that we name here *expansion*. Logical conflicts take place if concurrent operations overlap: because of the lock-free nature of the algorithm, several concurrent operations can run simultaneously, but usually only one retry can logically succeed. We show that the additional work produced by the failures is not necessarily harmful for the system-wise performance.

We then show how throughput can be computed by connecting these two key factors in an iterative way. We start by estimating the expansion probabilistically, and emulate the effect of stall time introduced by the hardware conflicts as extra work added to each thread. Then we estimate the number of failed operations, that in turn lead to additional extra work, by computing again the expansion on a system setting where those two new amounts of work have been incorporated, and reiterate the process; the convergence is ensured by a fixed-point search.

We consider the class of lock-free algorithms that can be modeled as a linear composition of fixed size retry loops. This class covers numerous extensively used lock-free designs such as stacks [16] (Pop, Push), queues [14] (Enqueue, Dequeue), counters [7] (Increment, Decrement) and priority queues [13] (DeleteMin).

To evaluate the accuracy of our model and analysis framework, we performed experiments both on synthetic tests, that capture a wide range of possible abstract algorithmic designs, and on several reference implementations of extensively studied lock-free data structures. Our evaluation results reveal that our model is able to capture the behavior of all the synthetic and real designs for all different numbers of threads and sizes of parallel work (consequently also contention). We also evaluate the use of our analysis as a tool for tuning the performance of lock-free code by selecting the appropriate back-off strategy that will maximize throughput by comparing our method against widely known back-off policies, namely linear and exponential.

The rest of the paper is organized as follows. We discuss related work in Section 2, then the problem is formally described in Section 3. We consider the logical conflicts in the absence of hardware conflicts in Section 4. In Section 5, we firstly show how to compute the expansion, then combine hardware and logical conflicts to obtain the final throughput estimate. We describe the experimental results in Section 6.

2 Related Work

Anderson *et al.* [3] evaluated the performance of lock-free objects in a single processor real-time system by emphasizing the impact of retry loop interference. Tasks can be preempted during the retry loop execution, which can lead to interference, and consequently to an inflation in retry loop execution due to retries. They obtained upper bounds for the number of interferences under various scheduling schemes for periodic real-time tasks.

Intel [11] conducted an empirical study to illustrate performance and scalability of locks. They showed that the critical section size, the time interval between releasing and re-acquiring the lock (that is similar to our parallel section size) and number of threads contending the lock are vital parameters.

Failed retries do not only lead to useless effort but also degrade the performance of successful ones by contending the shared resources. Alemany *et al.* [1] have pointed out this fact, that is in accordance with our two key factors, and, without trying to model it, have mitigated those effects by designing non-blocking algorithms with operating system support.

Alistarh *et al.* [2] have studied the same class of lock-free structures that we consider in this paper. The analysis is done in terms of scheduler steps, in a system where only one thread can be scheduled (and can then run) at each step. If compared with execution time, this is particularly appropriate to a system with a single processor and several threads, or to a system where the instructions of the threads cannot be done in parallel (*e.g.* multi-threaded program on a multi-core processor with only read and write on the same cache line of the shared memory). In our paper, the execution is evaluated in terms of processor cycles, strongly related to the execution time. In addition, the "parallel work" and the "critical work" can be done in parallel, and we only consider retry-loops with one Read and one CAS, which are serialized. In addition, they bound the asymptotic expected system latency (with a big O, when the number of threads tends to infinity), while in our paper we estimate the throughput (close to the inverse of system latency) for any number of threads.

3 Problem Statement

3.1 Running Program and Targeted Platform

In this paper, we aim at evaluating the throughput of a multi-threaded algorithm that is based on the utilization of a shared lock-free data structure. Such a

```
Procedure AbstractAlgorithm
1 Initialization();
2 while ! done do
3 |   Parallel_Work();
4 |   while ! success do
5 |   |   current ← Read(AP);
6 |   |   new ← Critical_Work(current);
7 |   |   success ← CAS(AP, current, new);
```

Fig. 1: Thread procedure

Fig. 2: Expansion

program can be abstracted by the Procedure AbstractAlgorithm (see Figure 1) that represents the skeleton of the function which is called by each spawned thread. It is decomposed in two main phases: the *parallel section*, represented on line 3, and the *retry loop*, from line 4 to line 7. A *retry* starts at line 5 and ends at line 7.

As for line 1, the function Initialization shall be seen as an abstraction of the delay between the spawns of the threads, that is expected not to be null, even when a barrier is used. We then consider that the threads begin at the exact same time, but have different initialization times.

The parallel section is the part of the code where the thread does not access the shared data structure; the work that is performed inside this parallel section can possibly depend on the value that has been read from the data structure, *e.g.* in the case of processing an element that has been dequeued from a FIFO (First-In-First-Out) queue.

In each retry, a thread tries to modify the data structure, and does not exit the retry loop until it has successfully modified the data structure. It does that by firstly reading the access point AP of the data structure, then according to the value that has been read, and possibly to other previous computations that occurred in the past, the thread prepares the new desired value as an access point of the data structure. Finally, it atomically tries to perform the change through a call to the *Compare-And-Swap* (*CAS*) primitive. If it succeeds, *i.e.* if the access point has not been changed by another thread between the first *Read* and the *CAS*, then it goes to the next parallel section, otherwise it repeats the process. The retry loop is composed of at least one retry, and we number the retries starting from 0, since the first iteration of the retry loop is actually not a retry, but a try.

The throughput of the lock-free algorithm, *i.e.* the number of successful data structure operations per unit of time, that we denote by T, is impacted by several parameters.

- *Algorithm parameters*: the amount of work inside a call to Parallel_Work (resp. Critical_Work) denoted by pw (resp. cw).
- *Platform parameters*: *Read* and *CAS* latencies (rc and cc respectively), and the number P of processing units (cores). We assume homogeneity for the latencies, *i.e.* every thread experiences the same latency when accessing an uncontended shared data, which is achieved in practice by pinning threads to the same socket.

3.2 Examples and Issues

We first present two straightforward upper bounds on the throughput, and describe the two kinds of conflict that keep the actual throughput away from those upper bounds.

3.2.1 Immediate Upper Bounds Trivially, the minimum amount of work $rlw^{(-)}$ in a given retry is $rlw^{(-)} = rc + cw + cc$, as we should pay at least the memory accesses and the critical work cw in between.

Thread-wise: A given thread can at most perform one successful retry every $pw + rlw^{(-)}$ units of time. In the best case, P threads can then lead to a throughput of $P/(pw + rlw^{(-)})$.

System-wise: By definition, two successful retries cannot overlap, hence we have at most 1 successful retry every $rlw^{(-)}$ units of time.

Altogether, the throughput T is upper bounded by the minimum of $1/(rc + cw + cc)$ and $P/(pw + rc + cw + cc)$, *i.e.*

$$T \leq \begin{cases} \frac{1}{rc+cw+cc} & \text{if } pw \leq (P-1)(rc+cw+cc) \\ \frac{P}{pw+rc+cw+cc} & \text{otherwise.} \end{cases} \quad (1)$$

3.2.2 Conflicts

Logical conflicts: Equation 1 expresses the fact that when pw is small enough, *i.e.* when $pw \leq (P-1)rlw^{(-)}$, we cannot expect that every thread performs a successful retry every $pw + rlw^{(-)}$ units of time, since it is more than what the retry loop can afford. As a result, some logical conflicts, hence unsuccessful retries, will be inevitable, while the others, if any, are called *wasted*.

Figure 3 depicts an execution, where the black parts are the calls to Initialization, the blue parts are the parallel sections, and the retries can be either unsuccessful — in red — or successful — in green. After the initial transient state, the execution contains actually, for each thread, one inevitable unsuccessful retry, and one wasted retry, because there exists a set of initialization times that lead to a cyclic execution with a single failure per thread and per period.

We can see on this example that a cyclic execution is reached after the transient behavior; actually, we show in Section 4 that, in the absence of hardware conflicts, every execution will become periodic, if the initialization times are spaced enough. In addition, we prove that the shortest period is such that, during this period, every thread succeeds exactly once. This finally leads us to define

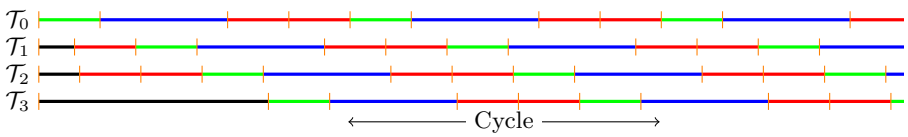

Fig. 3: Execution with one wasted retry, and one inevitable failure

the additional failures as wasted, since we can directly link the throughput with this number of wasted retries: a higher number of wasted retries implying a lower throughput.

Hardware conflicts: The requirement of atomicity compels the ownership of the data in an exclusive manner by the executing core. Therefore, overlapping parts of atomic instructions are serialized by the hardware, leading to stalls in subsequently issued ones. For our target lock-free algorithm, these stalls that we refer to as expansion become an important slowdown factor in case threads interfere in the retry loop. As illustrated in Figure 2, the latency for CAS can expand and cause remarkable decreases in throughput since the CAS of a successful thread is then expanded by others; for this reason, the amount of work inside a retry is not constant, but is, generally speaking, a function depending on the number of threads that are inside the retry loop.

3.2.3 Process We deal with the two kinds of conflicts separately and connect them together through the fixed-point iterative convergence.

In Section 5.1, we compute the expansion in execution time of a retry, noted e, by following a probabilistic approach. The estimation takes as input the expected number of threads inside the retry loop at any time, and returns the expected increase in the execution time of a retry due to the serialization of atomic primitives.

In Section 4, we are given a program without hardware conflicts described by the size of the parallel section $pw^{(+)}$ and the size of a retry $rlw^{(+)}$. We compute upper and lower bounds on the throughput T, the number of wasted retries w, and the average number of threads inside the retry loop P_{rl}. Without loss of generality, we can normalize those execution times by the execution time of a retry, and define the parallel section size as $pw^{(+)} = q + r$, where q is a non-negative integer and r is such that $0 \leq r < 1$. This pair (together with the number of threads P) constitutes the actual input of the estimation.

Finally, we combine those two outcomes in Section 5.2 by emulating expansion through work not prone to hardware conflicts and obtain the full estimation of the throughput.

4 Execution without hardware conflict

We show in this section that, in the absence of hardware conflicts, the execution becomes periodic, which eases the calculation of the throughput. We start by defining some useful concepts: (f, P)-cyclic executions are special kind of periodic executions such that within the shortest period, each thread performs exactly f unsuccessful retries and 1 successful retry. The *well-formed seed* is a set of events that allows us to detect an (f, P)-*cyclic execution* early, and the *gaps* are a measure of the quality of the synchronization between threads. The idea is to iteratively add threads into the game and show that the periodicity is maintained. Theorem 1 establishes a fundamental relation between gaps and

well-formed seeds, while Theorem 2 proves the periodicity, relying on the disjoint cases depicted on Figures 4a, 4b and 4c. We recall that the complete version of the proofs can be found in [5], together with additional Lemmas. Finally, we exhibit upper and lower bounds on throughput and number of failures, along with the average number of threads inside the retry loop.

4.1 Setting

In preamble, note that the events are strictly ordered (according to their instant of occurrence, with the thread id as a tie-breaker). As for correctness, *i.e.* to decide for the success or the failure of a retry, we need instants of occurrence for *Read* and *CAS*; we consider that the entrance (resp. exit) time of a retry is the instant of occurrence of the *Read* (resp. *CAS*).

4.1.1 Notations and Definitions We recall that P threads are executing the pseudo-code described in Procedure AbstractAlgorithm, one retry is of unit-size, and the parallel section is of size $pw^{(+)} = q + r$, where q is a non-negative integer and r is such that $0 \leq r < 1$. Considering a thread $\mathcal{T}_n$ which succeeds at time S_n; this thread completes a whole retry in 1 unit of time, then executes the parallel section of size $pw^{(+)}$, and attempts to perform again the operation every unit of time, until one of the attempt is successful.

Definition 1. *An execution with P threads is called (C, P)-cyclic execution if and only if (i) the execution is periodic, i.e. at every time, every thread is in the same state as one period before, (ii) the shortest period contains exactly one successful attempt per thread, (iii) the shortest period is $1 + q + r + C$.*

Definition 2. *Let $\mathcal{S} = (\mathcal{T}_i, S_i)_{i \in [\![0, P-1]\!]}$, where $\mathcal{T}_i$ are threads and S_i ordered times, i.e. such that $S_0 < \cdots < S_{P-1}$. $\mathcal{S}$ is a seed if and only if for all $i \in [\![0, P-1]\!]$, $\mathcal{T}_i$ does not succeed between S_0 and S_i, and starts a retry at S_i.*

We define $f(\mathcal{S})$ as the smallest non-negative integer such that $S_0 + 1 + q + r + f(\mathcal{S}) > S_{P-1} + 1$, i.e. $f(\mathcal{S}) = \max(0, \lceil S_{P-1} - S_0 - q - r \rceil)$. When $\mathcal{S}$ is clear from the context, we denote $f(\mathcal{S})$ by f.

Definition 3. *$\mathcal{S}$ is a well-formed seed if and only if for each $i \in [\![0, P-1]\!]$, the execution of thread $\mathcal{T}_i$ contains the following sequence: a successful retry starting at S_i, the parallel section, f unsuccessful retries, then a successful retry.*

Those definitions are coupled through the two natural following properties:

Property 1. Given a (C, P)-cyclic execution, any seed $\mathcal{S}$ including P consecutive successes is a well-formed seed, with $f(\mathcal{S}) = C$.

Property 2. If there exists a well-formed seed in an execution, then after each thread succeeded once, the execution coincides with an (f, P)-cyclic execution.

Together with the seed concept, we define the notion of *gap*. The general idea of those gaps is that within an (f, P)-cyclic execution, the period is higher than $P \times 1$, which is the total execution time of all the successful retries within the period. The difference between the period (that lasts $1 + q + r + f$) and P, reduced by r (so that we obtain an integer), is referred as *lagging time* in the following. If the threads are numbered according to their order of success (modulo P), as the time elapsed between the successes of two given consecutive threads is constant (during the next period, this time will remain the same), this lagging time can be seen in a circular manner: the threads are represented on a circle whose length is the lagging time increased by r, and the length between two consecutive threads is the time between the end of the successful retry of the first thread and the start of the successful retry of the second one. More formally, for all $(n, k) \in [\![0, P-1]\!]^2$, we define the gap $G_n^{(k)}$ between $\mathcal{T}_n$ and its k^{th} predecessor based on the gap with the first predecessor:

$$\begin{cases} \forall n \in [\![1, P-1]\!] \quad ; \quad G_n^{(1)} = S_n - S_{n-1} - 1 \\ G_0^{(1)} = S_0 + q + r + f - S_{P-1} \end{cases},$$

which leads to the definition of higher order gaps: $\forall n \in [\![0, P-1]\!]; \forall k > 0; G_n^{(k)} = \sum_{j=n-k+1}^{n} G_{j \bmod P}^{(1)}$.

For consistency, for all $n \in [\![0, P-1]\!]$, $G_n^{(0)} = 0$.

Equally, the gaps can be obtained from the successes: for all $k \in [\![1, P-1]\!]$,

$$G_n^{(k)} = \begin{cases} S_n - S_{n-k} - k & \text{if } n > k \\ S_n - S_{P+n-k} + 1 + q + r + f - k & \text{otherwise} \end{cases} \tag{2}$$

Note that, in an (f, P)-cyclic execution, the lagging time is the sum of all first order gaps, reduced by r.

4.2 Cyclic Executions

We only give the two main theorems used to show the existence of cyclic executions. The details can be found in the companion research report [5].

Theorem 1. *Given a seed $\mathcal{S} = (\mathcal{T}_i, S_i)_{i \in [\![0, P-1]\!]}$, $\mathcal{S}$ is a well-formed seed if and only if for all $n \in [\![0, P-1]\!]$, $0 \leq G_n^{(f)} < 1$.*

Theorem 2. *Assuming $r \neq 0$, if a new thread is added to an $(f, P-1)$-cyclic execution, then all the threads will eventually form either an (f, P)-cyclic execution, or an $(f+1, P)$-cyclic execution.*

Proof. We decompose the Theorem into three Lemmas which we describe here graphically:

- If all gaps of $(f+1)^{\text{th}}$ order are less than 1, then every existing thread will fail once more, and the new steady-state is reached immediately. See Figure 4a.
- Otherwise:

- Either: everyone succeeds once, whereupon a new (f, P)-cyclic execution is formed. See Figure 4b.
- Or: before everyone succeeds again, a new (f, P')-cyclic execution, where $P' \leq P$, is formed, which finally leads to an (f, P)-cyclic execution. See Figure 4c. □

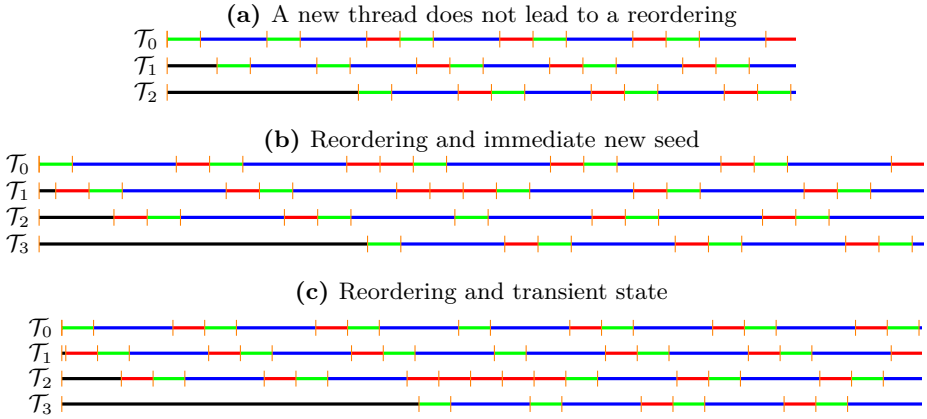

Fig. 4: Illustration of Theorem 2

4.3 Throughput Bounds

The periodicity offers an easy way to compute the expected number of threads inside the retry loop, and to bound the number of failures and the throughput.

Lemma 1. *In an (f, P)-cyclic execution, the throughput is $T = \frac{P}{q+r+1+f}$, and the average number of threads in the retry loop $P_{rl} = P \times \frac{f+1}{q+r+f+1}$.*

Lemma 2. *The number of failures is tighly bounded by $f^{(-)} \leq f \leq f^{(+)}$, and throughput by $T^{(-)} \leq T \leq T^{(+)}$, where*

$$f^{(-)} = \begin{cases} P-q-1 & \text{if } q \leq P-1 \\ 0 & \text{otherwise} \end{cases}, \qquad T^{(-)} = \begin{cases} \frac{P}{P+r} & \text{if } q \leq P-1 \\ \frac{P}{q+r+1} & \text{otherwise.} \end{cases}$$

$$f^{(+)} = \left\lfloor \frac{1}{2}\left((P-1-q-r) + \sqrt{(P-1-q-r)^2+4P}\right)\right\rfloor, \; T^{(+)} = \frac{P}{q+r+1+f^{(+)}}.$$

5 Expansion and Complete Throughput Estimation

5.1 Expansion

Interference of threads does not only lead to logical conflicts but also to hardware conflicts which impact the performance significantly. We model the behavior of

the cache coherency protocols which determine the interaction of overlapping *Read*s and *CAS*s. By taking MESIF [9] as basis, we come up with the following assumptions. When executing an atomic *CAS*, the core gets the cache line in exclusive state and does not forward it to any other requesting core until the instruction is retired. Therefore, requests stall for the release of the cache line which implies serialization. On the other hand, ongoing *Read*s can overlap with other operations. As a result, a *CAS* introduces expansion only to overlapping *Read* and *CAS* operations that start after it, as illustrated in Figure 2.

Furthermore, we assume that *Read*s that are executed just after a *CAS* do not experience expansion (as the thread already owns of the data), which takes effect at the beginning of a retry following a failing attempt. Thus, read expansions need only to be considered before the 0^{th} retry. In this sense, read expansion can be moved to parallel section and calculated in the same way as *CAS* expansion is calculated.

To estimate expansion, we consider the delay that a thread can introduce, provided that there is already a given number of threads in the retry loop. The starting point of each *CAS* is a random variable which is distributed uniformly within an expanded retry. The cost function d provides the amount of delay that the additional thread introduces, depending on the point where the starting point of its *CAS* hits. By using this cost function we can formulate the expansion increase that each new thread introduces and derive the differential equation below to calculate the expansion of a *CAS*.

Lemma 3. *The expansion of a CAS operation is the solution of the following system of equations:*

$$\begin{cases} e'\left(P_{rl}\right) & = cc \times \dfrac{\frac{cc}{2} + e\left(P_{rl}\right)}{rc + cw + cc + e\left(P_{rl}\right)} \\ e\left(P_{rl}^{(0)}\right) & = 0 \end{cases}, \qquad \textit{where } P_{rl}^{(0)} \textit{ is the point where expansion begins.}$$

Proof. To prove the theorem, we compute $e\left(P_{rl}+h\right)$, where $h \leq 1$, by assuming that there are already P_{rl} threads in the retry loop, and that a new thread attempts to *CAS* during the retry, within a probability h: $e\left(P_{rl}+h\right) = e\left(P_{rl}\right) + h \times \int_0^{rlw^{(+)}} \frac{d(t)}{rlw^{(+)}}\, dt$. The complete proof appears in the companion research report [5].

5.2 Throughput Estimate

It remains to combine hardware and logical conflicts in order to obtain the final upper and lower bounds on throughput. We are given as an input the expected number of threads P_{rl} inside the retry loop. We firstly compute the expansion accordingly, by solving numerically the differential equation of Lemma 3. As explained in the previous subsection, we have $pw^{(+)} = pw + e$, and $rlw^{(+)} = rc + cw + e + cc$. We can then compute q and r, that is the input set (together with the total number of threads P) of the method described in Section 4. Assuming that the initialization times of the threads are spaced enough, the execution will

superimpose an (f, P)-cyclic execution. Thanks to Lemma 1, we can compute the average number of threads inside the retry loop, that we note by $h_f(P_{rl})$. A posteriori, the solution is consistent if this average number of threads inside the retry loop $h_f(P_{rl})$ is equal to the expected number of threads P_{rl} that has been given as an input.

Several (f, P)-cyclic executions belong to the domain of the possible outcomes, but we are interested in upper and lower bounds on the number of failures f. We can compute them through Lemma 2, along with their corresponding throughput and average number of threads inside the retry loop. We note by $h^{(+)}(P_{rl})$ and $h^{(-)}(P_{rl})$ the average number of threads for the lowest number of failures and highest one, respectively. Our aim is finally to find $P_{rl}^{(-)}$ and $P_{rl}^{(+)}$, such that $h^{(+)}(P_{rl}^{(+)}) = P_{rl}^{(+)}$ and $h^{(-)}(P_{rl}^{(-)}) = P_{rl}^{(-)}$. If several solutions exist, then we want to keep the smallest, since the retry loop stops to expand when a stable state is reached.

Note that we also need to provide the point where the expansion begins. It begins when we start to have failures, while reducing the parallel section. Thus this point is $(2P-1)rlw^{(-)}$ (resp. $(P-1)rlw^{(-)}$) for the lower (resp. upper) bound on the throughput.

Theorem 3. *Let (x_n) be the sequence defined recursively by $x_0 = 0$ and $x_{n+1} = h^{(+)}(x_n)$. If $pw \geq rc + cw + cc$, then $P_{rl}^{(+)} = \lim_{n \to +\infty} x_n$.*

Proof. In [5], we prove that $h^{(+)}$ is non-decreasing when $pw \geq rc + cw + cc$, and obtain the above theorem by applying the Theorem of Knaster-Tarski.

The same line of reasoning holds for $h^{(-)}$. We point out that when $pw < rlw^{(-)}$, we scan the interval of solution, and have no guarantees about the fact that the solution is the smallest one; still this corresponds to very extreme cases.

6 Experimental Evaluation

We validate our model and analysis framework through successive steps, from synthetic tests, capturing a wide range of possible abstract algorithmic designs, to several reference implementations of extensively studied lock-free data structure designs that include cases with non-constant parallel section and retry loop. The complete results can be found in [5] and the numerical simulation code in [4].

6.1 Setting

We have conducted experiments on an Intel ccNUMA workstation system. The system is composed of two sockets, that is equipped with Intel Xeon E5-2687W v2 CPUs. In a socket, the ring interconnect provides L3 cache accesses and core-to-core communication. Threads are pinned to a single socket to minimize non-uniformity in *Read* and *CAS* latencies. Due to the bi-directionality of the ring that interconnects L3 caches, uncontended latencies for intra-socket communication between cores do not show significant variability. The methodology

in [6] is used to measure the *CAS* and *Read* latencies, while the work inside the parallel section is implemented by a for-loop of *Pause* instructions.

In all figures, y-axis provides the throughput, while the parallel work is represented in x-axis in cycles. The graphs contain the high and low estimates (see Section 4), corresponding to the lower and upper bound on the wasted retries, respectively, and an additional curve that shows the average of them.

6.2 Synthetic Tests

For the evaluation of our model, we first create synthetic tests that emulate different design patterns of lock-free data structures (value of cw) and different application contexts (value of pw).

Generally speaking, in Figure 5, we observe two main behaviors: when pw is high, the data structure is not contended, and threads can operate without failure (unsuccessful retries). When pw is low, the data structure is contended, and depending on the size of cw (that drives the expansion) a steep decrease in throughput or just a roughly constant bound on the performance is observed.

An interesting fact is the waves appearing on the experimental curve, especially when the number of threads is low or the critical work big. This behavior is originating because of the variation of r with the change of parallel work, a fact that is captured by our analysis.

6.3 Treiber's Stack

The lock-free stack by Treiber [16] is typically the first example that is used to validate a freshly-built model on lock-free data structures. A Pop contains a retry loop that first reads the top pointer and gets the next pointer of the element to obtain the address of the second element in the stack, before attempting to

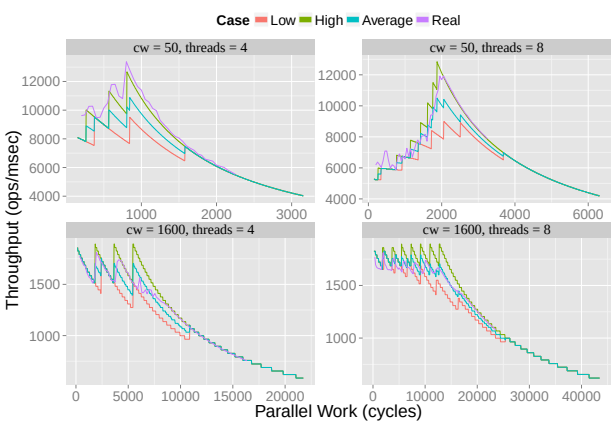

Fig. 5: Synthetic program

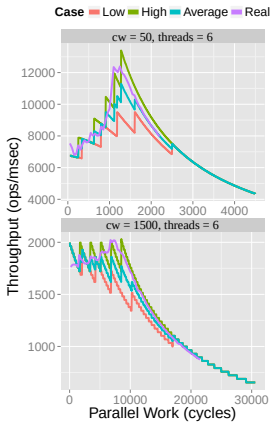

Fig. 6: Pop on stack

CAS with the address of the second element. The access to the next pointer of the first element occurs between the *Read* and the *CAS*. Thus, it represents the work in *cw*. By varying the number of elements that are popped at the same time, and the cache misses implied by the reads, we vary the work in *cw* and obtain the results depicted in Figure 6.

6.4 Discussion

In this subsection we discuss the adequacy of our model, specifically the cyclic argument, to capture the behavior that we observe in practice. Figure 7 illustrates the frequency of occurrence of a given number of consecutive fails, together with average fails per success values and the throughput values, normalized by a constant factor so that they can be seen on the graph. In the background, the frequency of occurrence of a given number of consecutive fails before success is presented. As a remark, the frequency of 6+ fails is plotted together with 6. We expect to see a frequency distribution concentrated around the average fails per success value, within the bounds computed by our model.

While comparing the distribution of failures with the throughput, we could conjecture that the bumps come from the fact that the failures spread out. However, our model captures correctly the throughput variations and thus strips down the right impacting factor. The spread of the distribution of failures indicates the violation of a stable cyclic execution (that takes place in our model), but in these regions, r actually gets close to 0, as well as the minimum of all gaps. The scattering in failures shows that, during the execution, a thread is overtaken by another one. Still, as gaps are close to 0, the imaginary execution, in which we switch the two thread IDs, would create almost the same performance effect. This reasoning is strengthened by the fact that the actual average number of failures follows the step behavior, predicted by our model. This shows

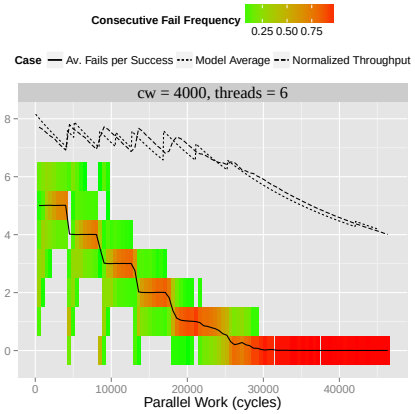

Fig. 7: Consecutive Fails Frequency

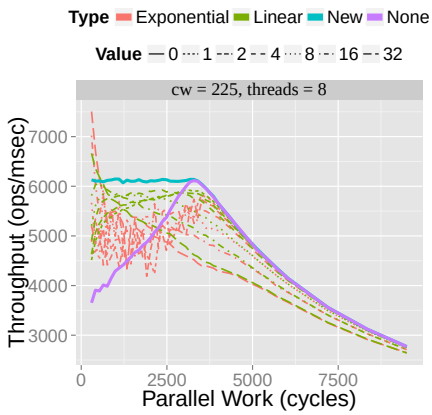

Fig. 8: Comparison of back-off schemes

that even when the real execution is not cyclic and the distribution of failures is not concentrated, our model that results in a cyclic execution remains a close approximation of the actual execution.

6.5 Back-Off Tuning

Together with our analysis comes a natural back-off strategy: we estimate the pw corresponding to the peak point of the average curve, and when the parallel section is smaller than the corresponding pw, we add a back-off in the parallel section, so that the new parallel section is at the peak point.

We have applied exponential, linear and our back-off strategy to the Enqueue/Dequeue experiment specified in [5] (sequence of Enqueue and Dequeue interleaved with parallel sections). Our back-off estimate provides good results for both types of distribution. In Figure 8 (where the values of back-off are steps of 115 cycles), the comparison is plotted for the Poisson distribution, which is likely to be the worst for our back-off. Our back-off strategy is better than the other, except for very small parallel sections, but the other back-off strategies should be tuned for each value of pw.

7 Conclusion

In this paper, we have modeled and analyzed the performance of a general class of lock-free algorithms, and have so been able to predict the throughput of such algorithms, on actual system executions. The analysis rely on the estimation of two impacting factors that lower the throughput: on the one hand, the expansion, due to the serialization of the atomic primitives that take place in the retry loops; on the other hand, the wasted retries, due to a non-optimal synchronization between the running threads. We have derived methods to calculate those parameters, along with the final throughput estimate, that is calculated from a combination of these two previous parameters. As a side result of our work, this accurate prediction enables the design of a back-off technique that performs better than other well-known techniques, namely linear and exponential back-offs.

As a future work, we envision to enlarge the domain of validity of the model, in order to cope with data structures whose operations do not have constant retry loop, as well as the framework, so that it includes more various access patterns. The fact that our results extend outside the model we consider allows us to be optimistic on impacting factors introduced in this work. Finally, we also foresee studying back-off techniques that would combine a back-off in the parallel section (for lower contention) and in the retry loops (for higher robustness).

References

1. Alemany, J., Felten, E.W.: Performance issues in non-blocking synchronization on shared-memory multiprocessors. In: Hutchinson, N.C. (ed.) Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC). pp. 125–134. ACM (1992)

2. Alistarh, D., Censor-Hillel, K., Shavit, N.: Are lock-free concurrent algorithms practically wait-free? In: Shmoys, D.B. (ed.) Symposium on Theory of Computing (STOC). pp. 714–723. ACM (June 2014)
3. Anderson, J.H., Ramamurthy, S., Jeffay, K.: Real-time computing with lock-free shared objects. ACM Transactions on Computer Systems (TOCS) 15(2), 134–165 (1997)
4. Atalar, A., Renaud-Goud, P.: Numerical simulation code, `http://excess-project.eu/simu.tar`
5. Atalar, A., Renaud-Goud, P., Tsigas, P.: Analyzing the performance of lock-free data structures: A conflict-based model. Tech. Rep. 2014:15, Chalmers University of Technology (January 2015), `http://arxiv.org/abs/1508.03566`
6. David, T., Guerraoui, R., Trigonakis, V.: Everything you always wanted to know about synchronization but were afraid to ask. In: Kaminsky, M., Dahlin, M. (eds.) Proceedings of the ACM Symposium on Operating Systems Principles (SOSP). pp. 33–48. ACM (November 2013)
7. Dice, D., Lev, Y., Moir, M.: Scalable statistics counters. In: Blelloch, G.E., Vöcking, B. (eds.) Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). pp. 43–52. ACM (July 2013)
8. Dragicevic, K., Bauer, D.: A survey of concurrent priority queue algorithms. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS). pp. 1–6 (April 2008)
9. Goodman, J.R., Hum, H.H.J.: Mesif: A two-hop cache coherency protocol for point-to-point interconnects. Tech. rep., University of Auckland (November 2009), `http://hdl.handle.net/2292/11594`
10. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. Journal of Parallel and Distributed Computing (JPDC) 70(1), 1–12 (2010)
11. Intel: Lock scaling analysis on Intel® Xeon® processors. Tech. Rep. 328878-001, Intel (April 2013)
12. Kogan, A., Herlihy, M.: The future(s) of shared data structures. In: Halldórsson, M.M., Dolev, S. (eds.) Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC). pp. 30–39. ACM (July 2014)
13. Lindén, J., Jonsson, B.: A skiplist-based concurrent priority queue with minimal memory contention. In: Baldoni, R., Nisse, N., van Steen, M. (eds.) Proceedings of the International Conference on Principle of Distributed Systems (OPODIS). Lecture Notes in Computer Science, vol. 8304, pp. 206–220. Springer (December 2013)
14. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Burns, J.E., Moses, Y. (eds.) Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC). pp. 267–275. ACM (May 1996)
15. Shavit, N., Lotan, I.: Skiplist-based concurrent priority queues. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS). pp. 263–268 (May 2000)
16. Treiber, R.K.: Systems programming: Coping with parallelism. International Business Machines Incorporated, Thomas J. Watson Research Center (1986)
17. Valois, J.D.: Implementing Lock-Free Queues. In: Proceedings of International Conference on Parallel and Distributed Systems (ICPADS). pp. 64–69 (December 1994)