How Lock-free Data Structures Perform in Dynamic Environments: Models and Analyses

Aras Atalar[1], Paul Renaud-Goud[2], and Philippas Tsigas[3]

1 **Chalmers University of Technology, Göteborg, Sweden**
`aaras@chalmers.se`
2 **Toulouse Institute of Computer Science Research, Toulouse, France**
`prenaud@irit.fr`
3 **Chalmers University of Technology, Göteborg, Sweden**
`tsigas@chalmers.se`

Abstract

In this paper we present two analytical frameworks for calculating the performance of lock-free data structures. Lock-free data structures are based on retry loops and are called by application-specific routines. In contrast to previous work, we consider in this paper lock-free data structures in dynamic environments. The size of each of the retry loops, and the size of the application routines invoked in between, are not constant but may change dynamically. The new frameworks follow two different approaches. The first framework, the simplest one, is based on queuing theory. It introduces an average-based approach that facilitates a more coarse-grained analysis, with the benefit of being ignorant of size distributions. Because of this independence from the distribution nature it covers a set of complicated designs. The second approach, instantiated with an exponential distribution for the size of the application routines, uses Markov chains, and is tighter because it constructs stochastically the execution, step by step.

Both frameworks provide a performance estimate which is close to what we observe in practice. We have validated our analysis on

(i) several fundamental lock-free data structures such as stacks, queues, deques and counters, some of them employing helping mechanisms, and
(ii) synthetic tests covering a wide range of possible lock-free designs.

We show the applicability of our results by introducing new back-off mechanisms, tested in application contexts, and by designing an efficient memory management scheme that typical lock-free algorithms can utilize.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases Lock-free, Data Structures, Parallel Computing, Performance, Modeling, Analysis

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.23

1 Introduction

During the last two decades, lock-free data structures have received a lot of attention in the literature, and have been accepted in industrial applications, *e.g.* in the Intel's Threading Building Blocks Framework [12], the Java concurrency package [20] and the Microsoft .NET Framework [18].

Naturally, the development of lock-free data structures was accompanied by studies on the performance of such data structures, in order to characterize their scalability. Having no guarantee on the execution time of an individual operation, the time complexity analyses of lock-free algorithms have turned towards amortized analyses. The so-called amortized analyses

are thus interested in the worst-case behavior over a sequence of operations, which can be seen as a worst-case bound on the average time per operation. In order to cover various contention environments, the time complexity of the algorithms is often parameterized by different contention measures, such as point [5], interval [1] or step [6] contention. Nonetheless these investigations are targeting worst-case asymptotic behaviors. There is a lack of analytical results in the literature capable of describing the execution of lock-free algorithms on top of a hardware platform, and providing predictions that are close to what is observed in practice. Asymptotic bounds are particularly useful to rank different algorithms, since they rely on a strong theoretical background, but the presence of potentially high constants might produce misleading results. Yet, an absolute prediction of the performance can be of great importance by constituting the first step for further optimizations.

The common measure of performance for data structures is throughput, defined as the number of operations on the data structure per unit of time. To this end, this performance measure is usually obtained by considering an algorithm that strings together a pure sequence of calls to an operation on the data structure. However, when used in a more realistic context, the calls to the operations are mixed with application-specific code (that we call here parallel work). For instance, in a work-stealing environment designed with deques, a thread basically runs one of the following actions: pushing a new-generated task in its deque, popping a task from a deque or executing a task. The modifications on the deques are thus interleaved with deque-independent work. There exist some papers that consider in their experiments local computations between calls to operations during their respective evaluations, but the amount of local computations follows a given distribution varying from paper to paper, *e.g.* constant [17], uniform [10], exponential [22].

In this work, we derive a general approach for unknown distributions of the size of the application-specific code, as well as a tighter method when it follows an exponential distribution.

As for modeling the data structure itself, we use as a basis the universal construction described by Herlihy in [11], where it is shown that any abstract data type can get such a lock-free implementation, which relies on one retry loop. Moreover, we have particularly focused our experiments on data structures that present a low level of disjoint-access parallelism [13] (stack, queue, shared counter, deque). Coming back to amortized analyses, the time complexity of an operation is often expressed as a contention-free time complexity added with a contention overhead. In this paper, we want to model and analyze the impact of contention. Loosely speaking, the data structures that exhibit low level of disjoint-access parallelism have lightweight operations (*i.e.* low contention-free complexity) and they are prone to high contention overheads. In contrast, the data structures that present high level of disjoint-access parallelism, or that employ contention alleviation techniques, provide heavyweight operations (*i.e.* high contention-free complexity) and behave differently, compared to the previous ones, under contention. Our analyses examine this trade-off and then facilitate conscious decisions in the data structures design and use.

We propose two different approaches that analyze the performance of such data structures. On the one hand, we derive an average-based approach invoking queuing theory, which provides the throughput of a lock-free algorithm without any knowledge about the distribution of the parallel work. This approach is flexible but allows only a coarse-grained analysis, and hence a partial knowledge of the contention that stresses the data structure. On the other hand, we exhibit a detailed picture of the execution of the algorithm when the parallel work is instantiated with an exponential distribution, through a second complementary approach. We prove that the multi-threaded execution follows a Markovian process and a Markov chain

analysis allows us to pursue and reconstruct the execution, and to compute a more accurate throughput.

We finally show several ways to use our analyses and we evaluate the validity of our ideas by experimental results. Those two analysis approaches give a good understanding of the phenomena that drive the performance of a lock-free data structure, at a high-level for the average-based approach, and at a detailed level for the constructive method. Moreover, our results provide a common framework to compare different implementations of a data structure, in a fair manner. We also emphasize that there exist several concrete paths to apply our analyses. To this end, based on the knowledge about the application at hand, we implement two back-off strategies. We show the applicability of these strategies by tuning a Delaunay triangulation application [9] and a streaming pipeline component which is fed with trade exchange workloads [19]. These experiments reveal the validity of our analyses in the application domain, under non-synthetic workloads and diverse access patterns. We confirm the benefits of our theoretical results by designing a new adaptive memory management mechanism for lock-free data structures in dynamic environments which surpasses the traditional scheme and which is such that the loss in performance, when compared to a static data structure without memory management, is largely leveraged. This memory management mechanism is based on the analyses presented in this paper.

2 Related Work

Alistarh *et al.* [2] have studied the same class of lock-free data structures that we consider in this paper. They show initially that the lock-free algorithms are statistically wait-free and going further they exhibit upper bounds on the performance. Their 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 where the instructions of the threads cannot be done in parallel (*e.g.* multi-threaded program on a multi-core processor with only writes 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. Also, in our paper we estimate the throughput (close to the inverse of system latency) for any number of threads.

Comparing to our previous work: In [3], we illustrate the performance impacting factors and the model we use to cover a subset of lock-free structures that we consider in this paper. In the former paper, the analysis is built upon properties that arise only when the sizes of the critical work and the parallel work are *constant*. There, we show that the execution is not memoryless due to the natural synchrony provided by the retry loops; at the end of the line, we prove that the execution is cyclic and use this property to bound the rate of failed retries.

Here, we provide two new approaches which serve different purposes. In the first approach, we relax the assumptions regarding the critical work and parallel work parameters, that we respectively use to model the data structure operations and the application specific code from which the data structure operations are called. The first approach relies on the expected values of the size of the critical work and the parallel work. This allows us to cover, compared to our previous analysis, more advanced lock-free data structure operations, see Section 6.3. Also, we can analyse the data structures running on a larger variety of application specific environments, thanks to the relaxed assumption on the size of the parallel work. The second approach provides a tight analysis when the parallel work follows an exponential distribution.

We can observe a significant decrease in the performance when the parallel work is initiated with exponential distribution in comparison to the cases where the parallel work is constant as in our previous work, see [4]. The tight analyses, in our previous work and the second approach presented in this paper, reveal for the first time an analytical understanding of this phenomenon.

This paper is complementary to the previous work, not only because of the difference in the analysis tools, the extensive set of data structures and the application specific environments that it considers but also because they together exhibit the impact of the size distributions of the parallel work on the performance of lock-free data structures.

3 Preliminaries

We describe in this section the structure of the algorithm that is covered by our model. We explain how to analyze the execution of an instance of such an algorithm when executed by several threads, by slicing this execution into a sequence of adjacent success periods, where a success period is an interval of time during which exactly one operation returns. Each of the success periods is further split into two by the first access to the data structure in the considered retry loop. This execution pattern reflects fundamental phases of both analyses, whose first steps and general direction are outlined at the end of the section.

3.1 System Settings

All threads call Procedure AbstractAlgorithm (see Figure 1) when they are spawned. So each thread follows a simple though expressive pattern: a sequence of calls to an operation on the data structure, interleaved with some parallel work during which the thread does not try to modify the data structure. For instance, it can represent a work-stealing algorithm, as described in the introduction.

The algorithm is decomposed in two main sections: the *parallel section*, represented on line 2, and the *retry loop* (which represents one operation on the shared data structure) from line 3 to line 6. A *retry* starts at line 4 and ends at line 6. The outer loop that goes from line 1 to line 6 is designated as the *work loop*. 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. The retry loop is composed of at least one retry (and the first iteration of the retry loop is strictly speaking not a retry, but a try).

We denote by cc the execution time of a CAS when the executing thread does not own the cache line in exclusive mode, in a setting where all threads share a last level cache. Typically, there exists a thread that touches the data between two requests of the same thread, therefore this cost is paid at every occurrence of a CAS. As for the *Read*s, rc holds for the execution time of a cache miss. When a thread executes a failed CAS, it immediately reads the same cache line (at the beginning of the next retry), so the cache line is not missing, and the execution time of the *Read* is considered as null. However, when the thread comes back from the parallel section, a cache miss is paid. To conclude with the parameters related to the platform, we use P cores, where the CAS (resp. the *Read*) latency is identical for all cores, *i.e.* cc (resp. rc) is constant.

The algorithm is parameterized by two execution times. In the general case, the execution time of an occurrence of the parallel section (application-specific section) is a random variable that follows an unknown probability distribution. In the same way, the execution time of the critical work (specific to a data structure) can vary while following an unknown probability distribution. The only provided information is the mean value of those two execution times:

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

Figure 1 Thread procedure.

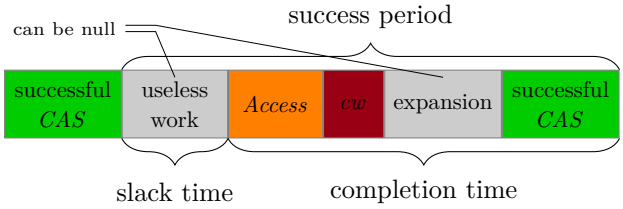

Figure 2 Success period.

cw for the critical work, and *pw* for the parallel work. These values will be given in units of work, where 1 u.o.w. = 50 cycles.

3.2 Execution Description

It has been underlined in [3] that there are two main conflicts that degrade the performance of the data structures which do not offer a great degree of disjoint-access parallelism: logical and hardware conflicts.

Logical conflicts occur when there are more than one thread in the retry loop at a given time (happens typically when the number of threads is high or when the parallel section is small). At any time, considering only the threads that are in the retry loop, there is indeed at most one thread whose retry will be successful (*i.e.* whose ending *CAS* will succeed), which implies the execution of more retries for the failing threads. In addition, after a thread executes successfully its final *CAS*, the other threads of the retry loop have first to finish their current retry before starting a potentially successful retry, since they are not informed yet that their current retry is doomed to failure. This creates some "holes" in the execution where all threads are executing useless work.

The threads will also experience *hardware conflicts*: if several threads are requesting the same data, so that they can operate a *CAS* on it, a single thread will be satisfied. All the other threads will have to wait until the current *CAS* is finished, and give a new try when this *CAS* is done. While waiting for the ownership of the cache line, the requesting threads cannot perform any useful work. This waiting time is referred to as *expansion*.

We now refine the description of the execution of the algorithm. The timeline is initially decomposed into a sequence of success periods that will define the throughput. A success period is an interval of time of the execution that

(i) starts after a successful *CAS*,
(ii) contains a single successful *CAS*,
(iii) finishes after this successful *CAS*.

To be successful in its retry, a thread has first to access the data structure, then modify it locally, and finally execute a *CAS*, while no other thread performs changes on the data structure. That is why each success period is further cut into two main phases (see Figure 2). During the first phase, whose duration is called the *slack time*, no thread is accessing the data structure. The second phase, characterized by the *completion time*, starts with the first access to the data structure (by any thread). Note that this *Access* could be either a *Read* (if the concerned thread just exited the parallel section) or a failed *CAS* (if the thread was already in the retry loop). The next successful *CAS* will come at least after *cw* (one thread has to traverse the critical work anyway), that is why we split the latter phase into: *cw*, then expansion, and finally a successful *CAS*.

3.3 Our Approaches

In this work, we propose two different approaches to compute the throughput of a lock-free algorithm, which we name as average-based and constructive. The average-based approach relies on queuing theory and is focused on the average behavior of the algorithm: the throughput is obtained through the computation of the expectation of the success period at a random time. As for the constructive approach, it describes precisely the instants of accesses and modifications to the data structure in each success period: in this way, we are able to deconstruct and reconstruct the execution, according to observed events. The constructive approach leads to a more accurate prediction at the expense of requiring more information about the algorithm: the distribution functions of the critical and parallel works have indeed to be instantiated.

In both cases, we partition the domain space into different levels of contention (or *modes*); these partitions are independent across approaches, even if we expect similarities, but in each case, cover the whole domain space (all values of critical work, parallel work and number of threads).

3.3.1 Average-based Analysis

We distinguish two main modes in which the algorithm can run: contended and non-contended. In the non-contended mode, *i.e.* when the parallel work is large or the number of threads is low, concurrent operations are not likely to collide. So every retry loop will count a single retry, and atomic primitives will not delay each other. In the contended mode, any operation is likely to experience unsuccessful retries before succeeding (logical conflicts), and a retry will last longer than in the non-contended mode because of the collision of atomic primitives (hardware conflicts).

Once all the parameters are given, the analysis is centered around the calculation of a single variable $\overline{P_{rl}}$, which represents the expectation of the number of threads inside the retry loop at a random instant. Based on this variable, we are able to express the expected expansion $\overline{e}\left(\overline{P_{rl}}\right)$ at a random time. As a next step, we show how this expansion can be used to estimate the expected slack time $\overline{st}\left(\overline{P_{rl}}\right)$ and the expected completion time $\overline{ct}\left(\overline{P_{rl}}\right)$, and at the end, the expected time of a success period $\overline{sp}\left(\overline{P_{rl}}\right)$.

3.3.2 Constructive Method

The previous average-based reasoning is founded on expected values at a random time, while in the constructive approach, we study each success period individually, based on the number of threads at the beginning of the considered success period. So we are able to exhibit more clearly the instants of occurrences of the different accesses and modifications to the data structure, and thus to predict the throughput more accurately.

We rely on the same set of values used in the average-based approach, but these values are now associated with a given success period. Thus the number of threads inside the retry loop P_{rl}, as well as the slack time and the completion time are evaluated at the beginning of each success period. We denote these times in the same way as in the first approach, but remove the bar on top since these values are not expectations any more.

The different contention modes do not characterize here the steady-state of the data structure as in the previous approach but are associated with the current success period. Accordingly, the contention can oscillate through different modes in the course of the execution. First, a success period is not contended when $P_{rl} = 0$, *i.e.* when there is no thread in the retry loop after a successful CAS. In this case, the first thread that exits the parallel section

will be successful, and the *Access* of the sequence will be a *Read*. Second, the contention of a success period is high when at any time during the success period, there exists a thread that is executing a *CAS*. In other words, at the end of each *CAS*, there is at least one thread that is waiting for the cache line to operate a *CAS* on it. This implies that the first access of the success period is a *CAS* and occurs immediately after the preceding successful *CAS*: the slack time is null. Third, the mid-contention mode takes place when $P_{rl} > 0$, while at the same time, there are not enough requesting threads to fill the whole success period with *CAS*'s (which implies a non-null slack time). Since these requesting threads have synchronized in the previous success period, *CAS*'s do not collide in the current success period, and because of that, the expansion is null.

4 Average-based Approach

We propose in this section our coarse-grained analysis to predict the performance of lock-free data structures. Our approach utilizes fundamental queuing theory techniques, describing the average behavior of the algorithm. In turn, we need only a minimal knowledge about the algorithm: the mean execution time values *cw* and *pw*. As explained in Section 3.3.1, the system runs in one of the two possible modes: either contended or uncontended.

4.1 Contended System

We first consider a system that is contended. When the system is contended, we use Little's law to obtain, at a random time, the expectation of the success period, which is the interval of time between the last and the next successful *CAS*'s (see Figure 2).

The stable system that we observe is the parallel section: threads are entering it (after exiting a successful retry loop) at an average rate, stay inside, then leave (while entering a new retry loop). The average number of threads inside the parallel section is $\overline{P_{ps}} = P - \overline{P_{rl}}$, each thread stays for an average duration of *pw*, and in average, one thread is exiting the retry loop every success period $\overline{sp}\left(\overline{P_{rl}}\right)$, by definition of the success period.

According to Little's law [14], we have:

$$\overline{P_{ps}} = pw \times 1/\overline{sp}\left(\overline{P_{rl}}\right), \text{ i.e. } \overline{sp}\left(\overline{P_{rl}}\right) = pw/(P - \overline{P_{rl}}). \tag{1}$$

We decompose a success period into two parts: slack time and completion time (as explained in Section 3.2). We express the expectation of the success period time as

$$\overline{sp}\left(\overline{P_{rl}}\right) = \overline{st}\left(\overline{P_{rl}}\right) + \overline{ct}\left(\overline{P_{rl}}\right). \tag{2}$$

When the data structure is contended, a thread is likely to be successful after some failed retries. Therefore a thread that is successful was already in the retry loop when the previous successful *CAS* occurred. The time before a thread starts its *Access* is then the time before a thread finishes its current critical work since there is a thread currently executing a *CAS*.

4.1.1 Expected Completion Time

Since the data structure is contended, numerous threads are inside the retry loop, and, due to hardware conflicts, a retry can experience expansion: the more threads inside the retry loop, the longer time between a *CAS* request and the actual execution of this *CAS*. The expectation of the completion time can be written as:

$$\overline{ct}\left(\overline{P_{rl}}\right) = cc + cw + \overline{e}\left(\overline{P_{rl}}\right) + cc, \tag{3}$$

where $\overline{e}\left(\overline{P_{rl}}\right)$ is the expectation of expansion when there are $\overline{P_{rl}}$ threads inside the retry loop, in expectation. This expansion can be computed in the same way as in [3], through the following differential equation:

$$\begin{cases} \overline{e}'\left(\overline{P_{rl}}\right) & = & cc \times \dfrac{cc/2 + \overline{e}\left(\overline{P_{rl}}\right)}{cc + cw + cc + \overline{e}\left(\overline{P_{rl}}\right)}, \\ \overline{e}\left(1\right) & = & 0 \end{cases}$$

by assuming that the expansion starts as soon as strictly more than 1 thread are in the retry loop, in expectation.

4.1.2 Expected Slack Time

Concerning the slack time, we consider that, at any time, the threads that are running the retry loop have the same probability to be anywhere in their current retry. However, when a thread is currently executing a CAS, the other threads cannot execute as well a CAS. The other threads are thus in their critical work or expansion. For every thread, the time before accessing the data structure is then uniformly distributed between 0 and $cw + \overline{e}\left(\overline{P_{rl}}\right)$. Using a well-known formula on the expectation of the minimum of uniformly distributed random variables, we show in [4] that:

$$\overline{st}\left(\overline{P_{rl}}\right) = \left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right)/(\overline{P_{rl}} + 1). \tag{4}$$

4.1.3 Expected Success Period

We just have to combine Equations 2, 3, and 4 to obtain the general expression of the expected success period under contention: $\overline{sp}\left(\overline{P_{rl}}\right) = \left(1 + 1/(\overline{P_{rl}} + 1)\right)\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) + 2cc$, which leads, according to Equation 1, to

$$\frac{1}{pw} \times \left(\frac{\overline{P_{rl}} + 2}{\overline{P_{rl}} + 1}\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) + 2cc\right) = \frac{1}{P - \overline{P_{rl}}}. \tag{5}$$

4.2 Non-contended System

When the system is not contended, logical conflicts are not likely to happen, hence each thread succeeds in its retry loop at its first *retry*. *A fortiori*, no hardware conflict occurs. Each thread still performs one success every work loop, and the success period is given by $\overline{sp}\left(\overline{P_{rl}}\right) = (pw + rc + cw + cc)/P$. Moreover, a thread spends in average pw units of time in the retry loop within each work loop. As this holds for every thread, we deduce $P - \overline{P_{rl}} = \overline{P_{ps}} = pw/(pw + rc + cw + cc) \times P$. Combining the two previous equations, we obtain

$$\frac{\overline{sp}\left(\overline{P_{rl}}\right)}{pw} = \frac{1}{P - \overline{P_{rl}}}, \text{ where } \overline{sp}\left(\overline{P_{rl}}\right) = \frac{rc + cw + cc}{\overline{P_{rl}}}. \tag{6}$$

4.3 Unified Solving

We have to decide whenever the data structure is under contention or not, and to find the corresponding solution. Concerning the frontier between contended and non-contended system, we can remark that Equations 5 and 6 are equivalent if and only if

$$\frac{rc + cw + cc}{\overline{P_{rl}}} = \frac{\overline{P_{rl}} + 2}{\overline{P_{rl}} + 1}\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) + 2cc, \tag{7}$$

which leads to Lemma 1.

▶ **Lemma 1.** *The system switches from being non-contended to being contended at* $\overline{P_{rl}} = P_{rl}^{(0)}$, *where*

$$P_{rl}^{(0)} = \frac{cc + cw - rc}{2(cw + 2cc)} \left(\sqrt{1 + \frac{4(rc + cw + cc)(cw + 2cc)}{(cc + cw - rc)^2}} - 1 \right).$$

Proof. The three following properties, proved in [4], demonstrate the lemma:
(i) $P_{rl}^{(0)}$ is the unique positive solution of Equation 7 if the expansion is set to 0,
(ii) $P_{rl}^{(0)} \leq 1$, and
(iii) there is no solution of Equation 7 with a non-null expansion. ◀

Thanks to Lemma 1, we can unify the success period as:

$$\overline{sp}\left(\overline{P_{rl}}\right) = \begin{cases} \left(rc + cw + cc\right) / \overline{P_{rl}} & \text{if } \overline{P_{rl}} \leq P_{rl}^{(0)} \\ \left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) \times \frac{\overline{P_{rl}+2}}{\overline{P_{rl}+1}} + 2cc & \text{otherwise.} \end{cases}$$

The unified success period obeys to the following equation

$$\overline{sp}\left(\overline{P_{rl}}\right) = \frac{pw}{P - \overline{P_{rl}}}. \tag{8}$$

We show in the following theorem how to compute the throughput estimate; the proof, presented in [4], manipulates equations in order to be able to use the fixed-point Knaster-Tarski theorem.

▶ **Theorem 2.** *The throughput can be obtained iteratively through a fixed-point search, as* $T = \left(\overline{sp}\left(\lim_{n\to+\infty} u_n\right)\right)^{-1}$, *where*

$$\begin{cases} u_0 = \frac{rc+cw+cc}{pw+rc+cw+cc} \times P \\ u_{n+1} = \frac{u_n \overline{sp}(u_n)}{pw+u_n \overline{sp}(u_n)} \times P \qquad \textit{for all } n \geq 0. \end{cases}$$

5 Constructive Approach

In this section, we instantiate the probability distribution of the parallel work with an exponential distribution. We have therefore a better knowledge of the behavior of the algorithm, particularly in medium contention cases, which allows us to follow a fine-grained approach that studies individually each successful operation together with every *CAS* occurrence. We provide an elegant and efficient solution that relies on a Markov chain analysis.

5.1 Process

We have seen in Section 3.3.2 that we split the contention domain into three modes: no contention, medium contention or high contention. We start from a configuration with a given number of threads P_{rl} after a successful *CAS*, and describe what will happen until the next successful *CAS*: what will be the mode of the next success period, and more precisely, which will be the number of threads at the beginning of the next success period.

As a basis, we consider the execution that would occur without any other thread exiting the parallel section (then entering the retry loop); we call this execution the *internal execution*. This execution follows the success period pattern described in Figure 2 (with an infinite slack time if the system is not contended). On top of this basic success period, we inject

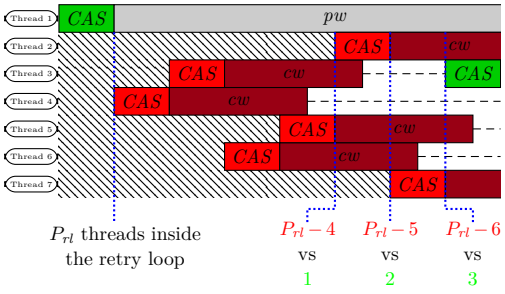

Figure 3 Highly-contended execution.

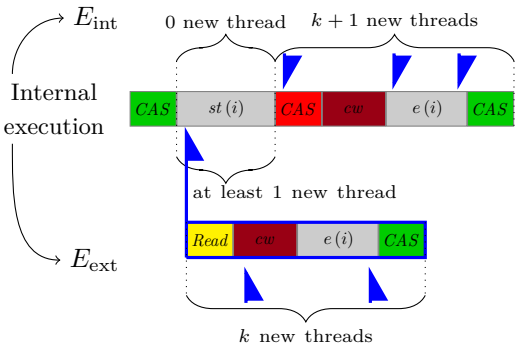

Figure 4 Possible executions.

the threads that can exit the parallel section, which has a double impact. On the one hand, they increase the number of threads inside the retry loop for the next success period. On the other hand, if the first thread that exits the parallel section starts its retry during the slack time of the success period of the internal execution, then this thread will succeed its *Access*, which is a *Read*, and will shrink the actual slack time of the current success period.

According to the distribution probability of the arrival of the new threads, we can compute the probability for the next success period to start with any number of threads. The expression of this stochastic sequence of success periods in terms of Markov chains results in the throughput estimate.

5.2 Expansion

The expansion, as before, represents the additional time in the execution time of a retry, due to the serialization of atomic primitives. However, in contrary to Section 4.1.1, we compute here this additional time in the current success period, according to the number of threads P_{rl} inside the retry loop at the beginning of the success period. The expansion only appears when the success period is highly contended, *i.e.* when we can find a continuous sequence of *CAS*'s all through the success period.

The expansion is highly correlated with the way the cache coherence protocol handles the exchange of cache lines between threads. We rely on the experiments of the research report associated with [2], which show that if several threads request for the same cache line in order to operate a *CAS*, while another thread is currently executing a *CAS*, they all have an equal probability to obtain the cache line when the current *CAS* is over.

We draw an illustrative example in Figure 3. The green *CAS*'s are successful while the red *CAS*'s fail. To lighten the picture, we hide what happened for the threads before they experience a failed *CAS*. The horizontal dash lines represent the time where a thread wants to access the data in order to operate a *CAS* but has to wait because another thread owns the data in exclusive mode. We can observe in this example that the first thread that accesses the data structure is not the thread whose operation returns.

We are given that P_{rl} threads are inside the retry loop at the end of the previous successful *CAS*, and we only consider those threads. When such a thread executes a *CAS* for the first time, this *CAS* is unsuccessful. The thread was in the retry loop when the successful *CAS* has been executed, so it has read a value that is not up-to-date anymore. However, this failed *CAS* will bring the current version of the value (to compare-and-swap) to the thread, a value that will be up-to-date until a successful *CAS* occurs.

So we have firstly a sequence of failed *CAS*'s until the first thread that operated its *CAS* within the current success period finishes its critical work. At this point, there exists a thread that is executing a *CAS*. When this *CAS* is finished, some threads compete to obtain the cache line. We have two bags of competing threads: in the first bag, the thread that just ended its critical work is alone, while in the second bag, there are all the threads that were in the retry loop at the beginning of the success period, and did not operate a *CAS* yet. The other, non-competing, threads are running their critical work and do not yet want to access the data.

As described before, every thread has the same probability to become the next owner of the cache line. If a thread from the first bag is drawn, then the *CAS* will be successful and the success period ends. Otherwise, the *CAS* is a failure, and we iterate at the end of this failed *CAS*. However, the thread that just failed its *CAS* is now executing its critical work, and does not request for a new *CAS* until this work has been done, thus it is not anymore in the second bag. In addition, the thread that had executed its *CAS* after the thread of the first bag is now back from its critical work and falls into the first bag. The process iterates until a thread is drawn from the first bag.

As a remark, note that we do not consider threads that are not in the retry loop at the beginning of the success period since even if they come back from the parallel section during the success period, their *Read* will be delayed and their *CAS* is likely to occur after the end of the success period.

Theorem 3, proved in [4], gives the explicit formula for the expansion.

▶ **Theorem 3.** *The expected time between the end of the critical work of the first thread that operates a CAS in the success period and the beginning of a successful CAS is given by:*

$$e\left(P_{rl}\right) = \lceil cw/cc \rceil cc - cw + \sum_{i=1}^{P_{com}} \frac{i(i-1)}{\left(P_{com}\right)^i} \frac{\left(P_{com}-1\right)!}{\left(P_{com}-i\right)!} \times cc, \quad \text{where } P_{com} = P_{rl} - \lceil cw/cc \rceil + 1.$$

5.3 Formalization

The parallel work follows an exponential distribution, whose mean is pw. More precisely, if a thread starts a parallel section at the instant t_1, the probability distribution of the execution time of the parallel section is $t \mapsto \lambda e^{-\lambda(t-t_1)} \mathbb{1}_{[t_1,+\infty[}(t)$, where $\lambda = 1/pw$. This probability distribution is memoryless, which implies that the threads that are executing their parallel section cannot be differentiated: at a given instant, the probability distribution of the remaining execution time is the same for all threads in the parallel section, regardless of when the parallel section began. For all threads, it is defined by: $t \mapsto \lambda \exp\left(-\lambda t\right)$, where $\lambda = 1/pw$.

For the behavior in the retry loop, we rely on the same approximation as in the previous section, *i.e.* when a successful thread exits its retry loop, the remaining execution time of the retry of every other thread that is still in the retry loop is uniformly distributed between 0 and the execution time of a whole retry. We have seen that the expectation of this remaining time is the size of the execution time of a retry divided by the number of threads inside the retry loop plus one. Here, we assume that a thread will start a retry at this time. This implies another kind of memoryless property: the behavior of a thread that is in the retry loop does not depend on the moment that it entered its retry loop.

To tackle the problem of estimating the throughput of such a system, we use an approach based on Markov chains. We study the behavior of the system over time, step by step: a state of the Markov chain represents the state of the system when the current success

period began (*i.e.* just after a successful *CAS*) and (thus) the system changes state at the end of every successful *CAS*. According to the current state, we are able to compute the probability to reach any other state at the beginning of the next success period. In addition, the two memoryless properties render the description of a state easy to achieve: the number of threads inside the retry loop when the current success begins, indeed fully characterizes the system.

We recall that P_{rl} is the number of threads inside the retry loop when the success period begins. The Markov chain is strongly related with P_{rl}, since it is composed of P states $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{P-1}$, where, for all $i \in [\![0, P-1]\!]$, the success period is in state $\mathcal{S}_i$ iff $P_{rl} = i$. For all $(i,j) \in [\![0, P-1]\!]^2$, $\mathbb{P}(\mathcal{S}_i \to \mathcal{S}_j)$ denotes the probability that a success characterized by $\mathcal{S}_j$ follows a success in state $\mathcal{S}_i$. $st(\mathcal{S}_i \to \mathcal{S}_j)$ denotes the slack time that passed while the system has gone from state $\mathcal{S}_i$ to state $\mathcal{S}_j$. This slack time can be expressed based on the slack time $st(i)$ of the internal execution, *i.e.* the execution that involves only the i threads of the retry loop and ignores the other threads (see Section 5.1). In the same way, we denote by $ct(i)$ the completion time of the internal execution, hence $ct(i) = cc + cw + e(i) + cc$.

We have seen that the level of contention (mode) is determined by P_{rl}, hence the interval $[\![0, P-1]\!]$ can be partitioned into $[\![0, P-1]\!] = \mathcal{I}_{\text{noc}} \cup \mathcal{I}_{\text{mid}} \cup \mathcal{I}_{\text{hi}}$, where the partitions correspond to the different contention levels. So, by definition, $\mathcal{I}_{\text{noc}} = \{0\}$, and for all $i \in \mathcal{I}_{\text{noc}} \cup \mathcal{I}_{\text{mid}}$, $e(i) = 0$ (see Section 3.3.2).

The success period is highly-contended, *i.e.* we have a continuous sequence of *CAS*'s in the success period, if the sum of the execution time of all the *CAS*'s that need to be operated exceeds the critical work. Hence $\mathcal{I}_{\text{hi}} = [\![i_{\text{hi}}, P-1]\!]$, where $i_{\text{hi}} = \min\{i \in [\![1, P-1]\!] \mid i \times cc > cw\}$. In addition, as the sequence of *CAS*'s is continuous when the contention is high, the slack time is null when the success period is highly contended, *i.e.*, for all $i \in \mathcal{I}_{\text{hi}}$, $st(i) = 0$, and *a fortiori*, $st(\mathcal{S}_i \to \mathcal{S}_\star) = 0$.

Otherwise, the success period is in medium contention, hence $\mathcal{I}_{\text{mid}} = [\![1, i_{\text{hi}} - 1]\!]$. Moreover, if $i \in \mathcal{I}_{\text{mid}}$, $st(i) > 0$, and $e(i) = 0$, because the *CAS*'s synchronized during the previous success period and will not collide any more in the current success period.

Everything is now in place to be able to obtain the stationary distribution of the Markov chain, and in turn to compute the throughput and the failure rate estimates. The reasoning that leads to the computation of the probability of going from state $\mathcal{S}_i$ to state $\mathcal{S}_{i+k}$ can be roughly summarized by Figure 4, where we start from an internal execution with i threads inside the retry loop and the blue arrows represent the threads that exit the parallel section. Two non-overlapping events can then potentially occur: either (event E_{ext}) the first thread exiting the parallel section starts within $[0, st(i)[$, *i.e.* in the slack time of the internal execution, or (event E_{int}) the first thread entering the retry loop starts after $t = st(i)$. The details can be found in [4].

6 Experiments

To validate our analysis results, we use two main types of lock-free algorithms. In the first place, we consider a set of basic algorithms where operations can be completed with a single successful *CAS*. This set of algorithms includes:

- **(i)** synthetic designs, that cover the design space of possible lock-free data structures;
- **(ii)** several fundamental designs of data structure operations such as lock-free stacks [21] (Pop, Push), queues [17] (Dequeue), counters [8] (Increment, Decrement).

As a second step, we consider more advanced lock-free operations that involve helping mechanisms, and show how to use our analysis in this context. Finally, in order to highlight

the benefits of the analysis framework, we show how it can be applied to

(i) determine a beneficial back-off strategy and

(ii) optimize the memory management scheme used by a data structure, in the context of an application.

We also give insights about the strengths of our two approaches. The constructive approach exhibits better predictions due to the tight estimation of the failing retries. On the other hand, the average-based approach is applicable to a broader spectrum of algorithmic designs as it leaves room to abstract complicated algorithmic designs.

6.1 Setting

We have conducted experiments on an Intel ccNUMA workstation system. The system is composed of two sockets 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. The methodology in [7] is used to measure the *CAS* and *Read* latencies, while the parallel work is implemented by a for-loop of *Pause* instructions. We show the experimental results with 8 threads.

In all figures, the y-axis shows both the throughput, *i.e.* number of operations completed per second, and the ratio of failing to successful retries (multiplied by 10^6, for readability), while the mean of the exponentially distributed parallel work pw is represented on the x-axis. The number of failures per success in the average-based approach is computed as $\overline{P_{rl}} - 1$ and in the constructive approach by stochastically counting the failing *CAS*'s inside a success period (see [4]).

We have also added a straightforward upper bound as a baseline approach, defined as the minimum of $1/(rc + cw + cc)$ (two successful retries cannot overlap) and $P/(pw + rc + cw + cc)$ (a thread can succeed only once in each work loop).

6.2 Basic Data Structures

Firstly, we consider lock-free operations that can be completed with a single successful *CAS*. We provide predictions, on the one hand, on a set of synthetic tests that have been constructed to abstract different possible design patterns of lock-free data structures (value of cw) and different application contexts (value of pw), and, on the other hand, on the well-known Treiber stack. The results, that show the satisfactory quality of the prediction, are depicted in [4].

6.3 Towards Advanced Data Structure Designs

Advanced lock-free operations generally require multiple pointer updates that cannot be done with a single *CAS*. One way to design such operations, in a lock-free manner, is to use helping mechanisms: an inconsistency will be fixed eventually by some thread. Here we consider two data structures that apply immediate helping, the queue from [17] and the deque designed in [15]. In the queue experiment (Figure 5), we run the Enqueue operation on the queue with and without memory management; in the deque experiment, each thread is dedicated to an end of the deque (equally distributed), while we vary the proportion of push operations (colors in Figure 6). More details about the implementations and the throughput estimate obtained through a slight modification of the average-based approach can be found in [4].

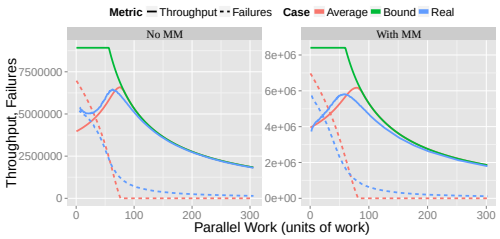

Figure 5 Enqueue on MS queue.

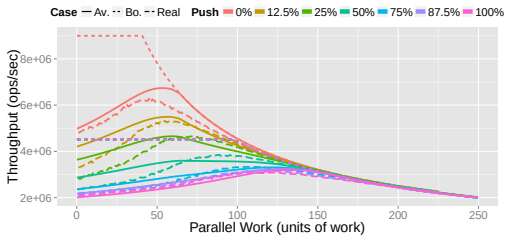

Figure 6 Operations on deque.

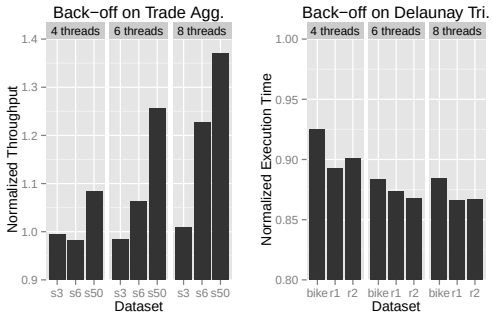

Figure 7 Performance impact of our back-off tunings.

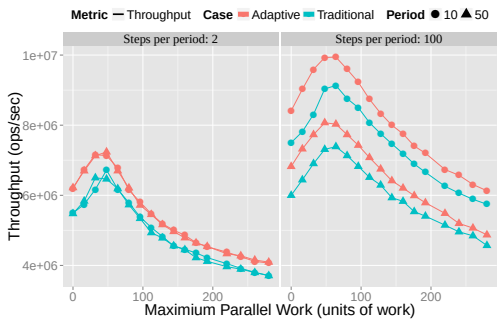

Figure 8 Adaptive MM with varying mean pw.

6.4 Applications

6.4.1 Back-off Optimizations

When the parallel work is known, we can deduce from our analysis a simple and efficient back-off strategy: as we are able to estimate the value for which the throughput is maximum, we just have to back-off for the time difference between the peak pw and the actual pw. In [4], we compare this back-off strategy against widely known strategies, namely exponential and linear, on a synthetic workload. In Figure 7, we apply our constant back-off on a Delaunay triangulation application [9], provided with several workloads. The application uses a stack in two phases, whose first phase pushes elements on top of the stack without delay. We are able to estimate a corresponding back-off time, and we plot the results by normalizing the execution time of our back-offed implementation with the execution time of the initial implementation.

A measure or an estimate of pw is not always available (and could change over time, see next section), therefore we propose also an adaptive strategy: we incorporate in the data structure a monitoring routine that tracks the number of failed retries, employing a sliding window. As our analysis computes an estimate of the number of failed retries as a function of pw, we are able to estimate the current pw, and hence the corresponding back-off time like previously.

We test our adaptive back-off mechanism on a workload originated from [19], where global operators of exchanges for financial markets gather data of trades with a microsecond accuracy. We assume that the data comes from several streams, each of them being associated with a thread. All threads enqueue the elements that they receive in a concurrent queue, so that they can be later aggregated. We extract from the original data a trade stream distribution that

we use to generate similar streams that reach the same thread; varying the number of streams to the same thread leads to different workloads. The results, represented as the normalized throughput (compared to the initial throughput) of trades that are enqueued when the adaptive back-off is used, are plotted in Figure 7. For any number of threads, the queue is not contended on workload s3, hence our improvement is either small or slightly negative. On the contrary, the workload s50 contends the queue and we achieve very significant improvement.

6.4.2 Memory Management Optimization

Memory Management (MM) is an inseparable part of dynamic concurrent data structures. In contrary to lock-based implementations, a node that has been *removed* from a lock-free data structure can still be accessed by other threads, *e.g.* if they have been delayed. Collective decisions are thus required in order to *reclaim* a node in a safe manner. A well-known solution to deal with this problem is the hazard pointers technique [16]. In an implementation of such design each thread lists the nodes that it accesses and the nodes that it has removed. When the number of nodes it has removed reaches a threshold, it reclaims its listed removed nodes that are not listed as accessed by any thread.

The main goal of our adaptive MM scheme is to distribute this extra-work in a way that the loss in performance is largely leveraged, knowing that additional work can be an advantage under high-contention (see previous section). The optimization is based on two main modifications. First, we divide the reclamation phase of the traditional MM scheme into quanta (equally-sized chunks), whose finer granularity allows for accurate back-off times. Second, we track continuously the contention level in the same way as our adaptive back-off. See [4].

We emulate the behavior of many scientific applications, that are built upon a pattern of alternating phases, that are communication-intensive (synchronization phase) or computation-intensive. Here we assume a synchronization ensured through a shared data structure, hence the communication-intensive phases correspond to a high access rate to the data structure, while the data structure is accessed at a low rate during a computation-intensive phase. The parallel work still follows an exponential distribution of mean pw, but pw varies in a sinusoidal manner with time. To study also the impact of the continuity of the change in pw, pw is set as a step approximation of a sine function. Thus, two additional parameters rule the experiment: the period of the oscillating function represents the length of the phases, and the number of steps within a period depicts how continuous the phase changes are.

In Figure 8, we compare our approach with the traditional implementation for different periods of the sine function, on the Dequeue of the Michael-Scott queue [17]. The adaptive MM, that relies on the analysis presented in this paper, outperforms the traditional MM because it provides an advantage both under low contention due to the costless (since delayed) invocation of the MM and under high contention due to the back-off effect.

7 Conclusion

In this paper we have presented two analyses for calculating the performance of lock-free data structures in dynamic environments. The first analysis has its roots in queuing theory, and gives the flexibility to cover a large spectrum of configurations. The second analysis makes use of Markov chains to exhibit a stochastic execution; it gives better results, but it is restricted to simpler data structures and exponentially distributed parallel work. We have shown how to use our results to tune applications using lock-free codes. These tuning methods include:

(i) the calculation of simple and efficient back-off strategies whose applicability is illustrated in application contexts;
(ii) a new adaptive memory management mechanism that acclimates to a changing environment.

The main differences between the data structures of this paper and linked lists, skip lists and trees occur when the size of the data structure grows. With large sizes, the performance is dominated by the traversal cost that is ruled by the cache parameters. The reduction in the size of the data structure decreases the traversal cost which in turn increases the probability of encountering an on-going *CAS* operation that delays the threads which traverse the link. The expansion, which can additionally be supported unfavorably by helping mechanisms, appears then as the main performance degrading factor. While the analysis becomes easier for high degrees of parallelism (large data structure size), being able to describe the behavior of lock-free data structures as the degree of parallelism changes constitutes the main challenge of our future work.

References

1 Yehuda Afek, Gideon Stupp, and Dan Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002. doi:10.1007/s004460100060.
2 Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? In *STOC*, pages 714–723. ACM, Jun 2014.
3 Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. Analyzing the performance of lock-free data structures: A conflict-based model. In *DISC*, pages 341–355, 2015.
4 Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. How lock-free data structures perform in dynamic environments: Models and analyses. Technical Report 2016:10, Chalmers University of Technology, Nov 2016. URL: http://arxiv.org/abs/1611.05793.
5 Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *JACM*, 50(4):444–468, 2003. doi:10.1145/792538.792541.
6 Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov. Computing with reads and writes in the absence of step contention. In *DISC*, pages 122–136, 2005. doi:10.1007/11561927_11.
7 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, pages 33–48. ACM, Nov 2013.
8 Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In *SPAA*, pages 43–52. ACM, Jul 2013.
9 Tanmay Gangwani, Adam Morrison, and Josep Torrellas. CASPAR: breaking serialization in lock-free multicore synchronization. In *ASPLOS*, pages 789–804, 2016.
10 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. *J. Par. Distr. Computing*, 70(1):1–12, 2010.
11 Maurice Herlihy. Wait-free synchronization. *TOPLAS*, 13(1):124–149, 1991.
12 Intel. Threading building blocks framework. Accessed: 2016-01-20. URL: https://www.threadingbuildingblocks.org/.
13 Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PoDC*, pages 151–160, 1994.
14 John D. C. Little. A proof for the queuing formula: L= λ w. *Operations research*, 9(3):383–387, 1961.
15 Maged M. Michael. Cas-based lock-free algorithm for shared deques. In *Euro-Par*, pages 651–660, 2003. doi:10.1007/978-3-540-45209-6_92.
16 Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE TPDS*, 15(8), Aug 2004.

17 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PoDC*, pages 267–275. ACM, May 1996.

18 Microsoft. NET Framework. Accessed: 2016-01-20. URL: `http://www.microsoft.com/net`.

19 NYSE. Daily trades from 2015-08-05. Accessed: 2016-05-05. URL: `http://www.nyxdata.com/Data-Products/Daily-TAQ#155`.

20 Oracle. Java concurrency package. Accessed: 2016-01-20. URL: `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`.

21 R. Kent Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

22 J. D. Valois. Implementing Lock-Free Queues. In *ICPADS*, pages 64–69, Dec 1994.