State of the art about non-blocking algorithms

Bilel SAGHROUCHNI
May 2020

***Abstract*—Non-blocking algorithms have allowed to spread concurrency, synchronization and performances through a wide kind of data structures. However, their development may be quite complex because of the need to know the operation of non-blocking algorithms, their implementation and a methodology to evaluate them.**
This work provides a presentation of the basis of synchronization through the different data structures we can find in the literature. To better understand the use of some of them, two non-blocking algorithms are compared thanks to a fine-grained approach to evaluate the design, use and performances of a non-blocking algorithm.

I. Introduction

The use of applications has been growing strongly in recent years [4] [3] and thus generates a very large amount of data: we are now in the age of Big Data. In addition to this quantitative explosion, the now common use of social networks, video streaming, or other network-based services, involves large amounts of data being generated and an increasing amount of traffic. This change in behavior requires efficient data structures that allow applications to access the different data they need to continuously read or modify it.

There are several approaches to access the data structure [5]. We can access it sequentially, but we will not be immune to latency and performance loss within our application. So it seems interesting to look at a concurrent access allowing several threads, for example, to access, read and modify the data structure at the same time. This approach is common to most of today's applications running on multi-core systems [1]. Thus thread synchronization is essential to avoid competitive problems and to ensure the smooth running of an application.
Many tools can be used for this, but most of them rely on the implementation of a lock applied to the shared data, so that full access to it is only guaranteed to one thread at a time. A dependency between threads is then established, which can compromise the completion of the task performed by the threads.

It is then interesting to address a partial lock and find a non-blocking solution to ensure thread synchronization and thus guarantee the completion of the current tasks within an acceptable time. This solution is not limited to the simple use of Mutexs, which would be inefficient for such data structures, but has the implementation of an algorithm that works, in the end, as a more or less strong lock. The primary objective of this state of the art is, after quickly introducing the basics of synchronization, to compare two implementations of non-blocking algorithms and to explain a methodology to evaluate their performances.

II. Related work

In this section we present the main principles of synchronization and then present two non-blocking algorithms.

A. Synchronization basis [5]

A concurrent data structure can be used only if all the threads which have access to it are synchronized. To do that efficiently, most of multi-cores systems use and support several synchronization primitives which allow to do atomic operation and so, avoid conflicts. These ones can be classified by their power or number of consensus which reflects the number of threads or processes among which a primitive can achieve a consensus. Another way to classify these synchronization primitives is by their scalability or combinability. If several memory requests to the same memory location can be combined in only one, the primitive is combinable. For instance, the CAS primitive has a number of infinite consensuses but is not combinable because it depends on the current value of the memory location.

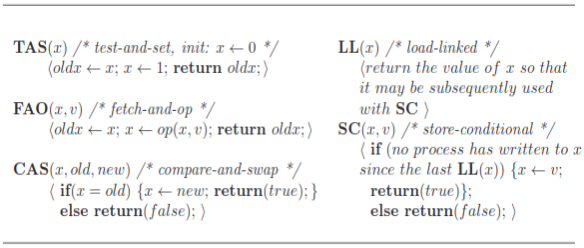

Figure 1 : Synchronization primitives [5]

There is a lot of different lock-free data structures and their use are motivated by what they intend to implement. We can differentiate lock-free data structures by their time and space complexity, the range of operation they support, their scalability, dynamic capacity, reliability and their compatibility and dependencies.

Developers can implement different data structures we can find in the literature as lock-free data structures. Stack, queue, list or hashtable could be lock-free using linked-list, the CAS primitive or circular array. Tree and dictionary can also be lock-free using respectively the LL/SC atomic primitive and an hash-table.

The problem in a concurrent data structure is to keep track of the free memory available and safely reclaim this allocated memory. So it is meaningful to use a good memory allocator, which ensures that an application keeps its memory blocks allocated, and chooses a good reclamation scheme to manage memory dynamically.

B. Non-blocking algorithm for distributed data structures [6]

When we want to write in memory, we use bounded physical addresses which correspond to physical memory (RAM) in our computer. Some applications may use most of the physical addresses, therefore to address memory more broadly, we look at logical address space which is very large and which applications can use. In this case, the system will be in charge of mapping the logical addresses with the physical ones. It allows the OS to keep control over its memory that would not be the case if all applications could directly write and modify the memory [2]. The challenge is to make the Partitioned Global Address Space (PGAS) non-blocking as it possible for shared memory. The PGAS provide remote-direct memory access (RDMA) operations like PUT and GET, which allow to write and read in the memory without the intervention of the CPU. In addition to the problem of concurrent safe memory reclaim, some programming languages like Chapel lacks support for atomic operation on an object which is the key for a non-blocking algorithm.

The GlobalAtomicObject may be a first solution. It performs a pointer compression and so benefits from the fact that processors use the lowest 48-bits for the virtual address and so 16-bits are available to local information. That is perfect to use atomic operations over the network with Chapel which only supports atomic operations up to 64-bits over the network, unlike to the LocalAtomicObject which is not a relevant option in a shared memory context. Moreover, AtomicObject provides 64 additional bits which handle the ABA problem and provide ABA-free atomic operations.
Concurrent-safe memory reclamation is at the basis of a non-blocking algorithm, and Epoch-Based reclamation provides it by using a system that utilizes epoch to determine the quiescence of object and learn when they are safe to be reclaimed [6]. Each thread which wants to access the data has to enter in an epoch before using the data and leave it afterwards. If a thread is in none epoch, we consider it as quiescent. An object associated to an epoch can be removed if all the threads are quiescent or if there are all in the next epoch.
In this work, EpochManager is in charge of providing garbage collection mechanisms that could scale in distributed memory and so delete all the objects that are not used by another object. This mechanism is also a good way to handle the logic related to epochs using particularly tokens which will be given to a task before it could access to a data structure protected by the epoch-based reclamation, therefore the epoch in which a task is in is tracked. Obviously, EpochManager is lock-free and so allows the data-structure which use it to stay non-blocking.
The AtomicObject could be a solution to a language which lacks support atomic operations and scales linearly with the number of locales either in shared memory or in distributed memory.

C. A non-blocking unordered list algorithm [12]

As we saw earlier, linked-list are fundamental data structures that are used by their own and for other data structures. In this section we will introduce a wait-free implementation of an unordered linked-list which could be used to implement stacks or hashtables. Lock-free lists already exists in particular the first one created by Valois [11] who uses a technique in which auxiliary nodes en-coded in-progress operations. Then other implementations were created using a pointer marking technique [7], wait-free lookup [8] or the fast-path-slow-path methodology [9] we introduce later.
In order to create a wait-free unordered linked-list implementation, we have to introduce a lock-free unordered list algorithm which serves as a basis for our final implementation. The operation supported by the lists are INSERT(k), REMOVE(k), CONTAINS(k) are presented in *Figure 2*. The list is composed of Node Objects which contain a key-value, a next pointer to the following Node and a state field which is helpful to coordinate the different operations which will occur. Two more fields *tid* and *prev* are useful only by the wait-free implementation we introduce later. Obviously, a head pointer points to the first element of the list and the linked-list object is always mapped to an abstract set object.

```
datatype NODE
  key   : ℕ      // integer data field
  state : ℕ      // INS, REM, DAT, or INV
  next  : NODE   // pointer to the successor
  prev  : NODE   // pointer to the predecessor
  tid   : ℕ      // thread id of the creater

global variables
  head  : NODE   // initially nil

1  function INSERT(k : ℕ) : 𝔹
2    h ← new NODE(k, INS, nil, nil, threadid)
3    ENLIST(h)
4    b ← HELPINSERT(h, k)
5    if ¬CAS(&h.state, INS, (b? DAT : INV)) then
6      HELPREMOVE(h, k)
7      h.state ← INV
8    return b

9  function REMOVE(k : ℕ) : 𝔹
10   h ← new NODE(k, REM, nil, nil, threadid)
11   ENLIST(h)
12   b ← HELPREMOVE(h, k)
13   h.state ← INV
14   return b

15 function CONTAINS(k : ℕ) : 𝔹
16   curr ← head
17   while curr ≠ nil do
18     if curr.key = k then
19       s ← curr.state
20       if s ≠ INV then
21         return (s = INS) ∨ (s = DAT)
22     curr ← curr.next
23   return false

24 procedure ENLIST(h : NODE)
25   while true do
26     old ← head
27     h.next ← old
28     if CAS(&head, old, h) then
29       return
```

Figure 2 : A Lock-free Unordered List [12]

INSERT and REMOVE operations begin by placing a node with the state INS or REM and linking them to the head of the list thanks to an intermediate function ENLIST. Two last functions HelpInsert and HelpRemove determine if the insert or remove is possible by checking the states of the nodes

composing the current list. The majority of those functions repeatedly performs the CAS operation, especially ENLIST which attempts to change head to point to h. However, this operation may fail for an unbounded number of times, blocking the thread in this function and so not be wait-free.

So the algorithm is lock-free because all the operations complete when any thread executes a bounded number of local steps, but we have to make it wait-free by making the ENLIST function wait-free. Using the enqueue technique introduced by Kogan and Petrank [8], it is possible to append a node at the tail of the list in a wait-free way. We now use the *prev* field because, in our case, nodes are appended at the head position.
Finally, the algorithm has to be adaptive because the wait-free approach imposes overhead in the common cases when contention is low. We employed the fast-path-slow-path methodology to achieve that. A thread starts by executing the fast path version of the ENLIST operation (the lock-free) and if it fails too many times, executes the low-path one (the wait-free). [9]
The FastPath lock-free list is always a constant factor slower than the lock-free unordered list, but the Adaptive algorithm remains close to FastPath.

III. Discussions

As noted until now, non-blocking algorithms are useful elements to guarantee completion of threads and performances, but their implementation can be different and so they may not have the same efficiency. Thus a developer has to be allowed to compare its non-blocking algorithm to another and evaluate its performances. Usually, to do that, we measured the number of operations completed in a period of time, but it could be deceptive depending on the properties of the object on which we use the algorithm (its size, complexity).
An approach using software and hardware metrics [10] provides developers with a meaningful way to evaluate a non-blocking algorithm. A methodology, which allows the developers to simulate an application's use of a non-blocking algorithm and draw from the metrics an accurate analysis, supports this approach. The software metrics are the number of times a thread tries to recruit another thread to complete its operation, number of times a thread has to remove another or number of times a thread has been recruited to complete an operation. Concerning the hardware metrics, number of clock cycles, number of CPU instructions or number of misaligned data access are used. This fine-grained methodology permits to provide information as the impact of the threads interference on the CPU. These two metrics are the basic blocks of the methodology presented in *Figure 3*.
Overall, these metrics treat of three main criterias : concurrency, scalability, compatibility and dependencies. Let's use them to compare the two non-blocking algorithms [6] [12] and the one used in the methodology [10] presented above .

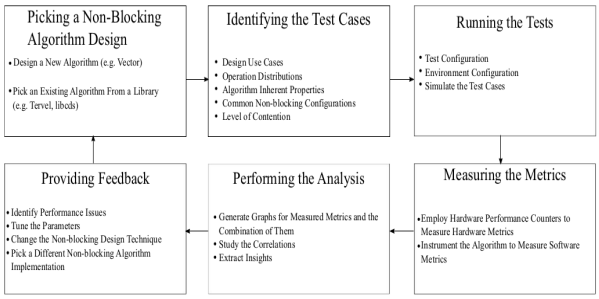

Figure 3 : Overview of the methodology [10]

Concerning concurrency, all the algorithms increase the number of threads but we can get different information : the throughput [12] or the number of operations completed [10]. As well as the increasing of threads we can expend the tasks they perform (read, write, compare and swap, pushback) [6]. Indeed this criteria is also relevant to evaluate the compatibility and dependencies and that is how we evaluate the EpochManager's performances [12] or the Tervel's vector ones [10]. The last algorithm, on the other hand, uses several benchmarks and different lists which do not use the same number of instructions to perform an INSERT or a REMOVE operation. Finally, the scalability is handled in the same way by increasing the number of threads or the number of tasks performed by these ones. We summarize the entire comparison in *Table 1* below.

Articles	[6]	[10]	[12]
Concurrency	• Atomic Object : 25% read, 25% write, 25% compare-and-swap, and 25% exchange operations • Epoche Manager : read-often write-rarely or read only	• Increasing of the number of threads and number of operations completed	• Increasing of the number of threads (observation of the throughput)
Scalability	• Atomic Object : increasing the number of tasks • Epoche Manager : increasing number of remote objects to be reclaimed	• Increasing the number of threads	• Increasing the number of threads
Compatibility and dependencies		• Compare the behavior of Tervel's vector across several use cases : different proportion of read and pushback	• Several benchmarks used : short list, WFList (uses 2 more CAS instructions), LFList (uses an extra CAS instr. in INSERT and one less in REMOVE)

Table 1 : Comparison of the three non-blocking algorithms

For all these algorithms, the tests are quite complete in a software way, but there is no variation in a hardware way. It would be interesting and would make more sense to use different processors or computers with different properties to evaluate the dependency criteria and check at the number of clock cycles for the concurrency one, for example.

IV. Conclusion

In this work, we provide a complete analysis of non-blocking algorithms. Their implementation needs a solid knowledge about data structures and synchronization, which are the building blocks for non-blocking algorithms. They allow to handle concurrent accesses and memory reclamation, whether within shared-memory or even distributed memory. Their complete analysis let developers know about precious information concerning its non-blocking algorithm and its use.

REFERENCES

[1] Abderazek Ben Abdallah. *Multicore Systems On-Chip: Practical Software/Hardware Design - Chapter 1 (Preview)*. Atlantis Press, Paris, 2 edition, 2013.

[2] George ALMASI. Pgas (partitioned global addressspace) languages. *Laser Spectroscopy IV: Proceedings of the Fourth International Conference*, 1979.

[3] App Annie. prévisions 2021 pour le marché des applications mobiles. *IPI Écoles*, 2017.

[4] App Annie. Prévisions app économie : 6000 milliards de $ de création de valeur. *App Annie*, 2017.

[5] Daniel Cederman, Anders Gidenstam, Phuong Ha, Hkan Sundell, Marina Papatriantafilou, and Philippas Tsigas. *Lock-Free Concurrent Data Structures*, chapter 3, pages 59–79. John Wiley Sons, Ltd, 2017.

[6] Garvit Dewan and Louis Jenkins. Paving the way for distributed non-blocking algorithms and data structures in the partitioned global address space, 2020.

[7] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, page 300–314, Berlin, Heidelberg, 2001. Springer-Verlag.

[8] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, page 223–234, New York, NY, USA, 2011. Association for Computing Machinery.

[9] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 141–150, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Damian Dechev Ramin Izadpanah, Steven Feldman. A methodology for performance analysis of non-blocking algorithms using hardware and software metrics. *IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*, May 2016.

[11] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, page 214–222, New York, NY, USA, 1995. Association for Computing Machinery.

[12] Kunlong Zhang, Yujiao Zhao, Yajun Yang, Yujie Liu, and Michael Spear. Practical non-blocking unordered lists. pages 239–253, 2013.