SAPIENZA
UNIVERSITÀ DI ROMA

School of Information Engineering, Computer Science and Statistics

---

Master of Science in Engineering in Computer Science

A Lock-Free O(1) Priority Queue For Pending Event Set Management

Advisor

Francesco Quaglia

Co-Advisor

Alessandro Pellegrini

External Reviewer

Andrea Vitaletti

Candidate

Romolo Marotta

January 2016

Academic Year 2014/2015

Romolo Marotta

A Lock-Free O(1) Priority Queue For Pending Event Set Management

Master's Thesis in Engineering in Computer Science

Sapienza Università di Roma
January 2016

Contents

List of Figures

Acknowledgements

My first word of gratitude goes to Alessandro Pellegrini, who has introduced me to the world of non-blocking algorithms and followed me during the developing of this thesis. Without his availability and serenity, this thesis would be much more difficult.

I would like to thank my advisor Prof. Francesco Quaglia that inspired me during his lessons. Thanks to him, I believe that very few things are impossible, but, if it comes to software, it is more likely that they are possible. Thus, no matter how hard it is, somewhere there must be a solution. This work would never come to light if he had not motived me to work so hard.

A word of graditude goes to Prof. Andrea Vitaletti that has revised this thesis.

I would like to thank Mauro Ianni for having analyzed my work with great humility and criticism.

A special thanks goes to Prof. Bruno Ciciani and Pierangelo Di Sanzo for hours spent with me and Andrea La Rizza. They have had the ability to make us feel their peers, but always teaching us something.

To Andrea La Rizza with whom I shared the last year of studies. Nights spent talking about computer security, seminars and philosophy have made me discover a valuable colleague and a great friend. I have spent most of the first year fighting agaist every homework with another friend, Fabio Angeletti. To Simone and Claudia reminding me that coffee is not a lunch. Another special thanks goes to Davide Tiriticco, invaluable friend, always available for an advice and a discussion on new projects. To Roberto, Daniele, Massimo, Silvia, Andrea and Luca for forcing me to leave the computer at home sometimes. To "Soliti" for being always present even after my long disappearances. To my bands, in particular Box 36 and New Carboneria: playing with you has been an honor.

I am very grateful to my parents for always believing in me and always encouraging me. I hope to give back even a fraction of what you were able to give me.

I thank all those I have not mentioned, but I think constantly, because we have shared something: a laugh, a sunset, a song.

To Giulia for being you with me.

Abstract

In Event-Driven programs the data-structure which contains not scheduled events is called Pending Event Set and is implemented as a priority queue. The literature offers several implementations that guarantee $O(1)$ time for enqueue and dequeue operations, but the wide diffusion of multicore processors imposes that Pending Event Set management has to be scalable as well as asymptotically efficient. In fact concurrent priority queues based on mutual exclusion limit the advantages of an increasing number of cores. To addressing this issue, several non-blocking priority queues have been proposed, but none targets constant time accesses, even if it is achieved by some blocking algorithms. On one hand there are algorithms that guarantee $O(1)$ access times, but they are based on locking primitives that reduce scalability, while on the other hand there are scalable non-blocking algorithms that have at least $O(\log n)$ asymptotic cost per access. Designing and implementing an efficient non-blocking priority queue with constant time access is still an open question. In this work we propose a practical lock-free implementation of a priority queue based on a multi-list object paired with an overflow data structure. Moreover in our performance evaluation, we show that our algorithm is more scalable than an efficient blocking implementation of the calendar queue.

The remainder of this thesis is organized as the following. In Chapter 1 we give an overview of concurrent programming, focusing in particular on non-blocking concurrent objects. Chapter 2 frame the problem of Pending Event Set management in the Event-Driven programming, outlining the state of the art. In Chapter 3 we present a deep description of our non-blocking priority queue, showing its operation and properties. The results of the experimental evaluation addressed to verify the validity of the proposed solution are reported in Chapter 4.

Chapter 1

Concurrent Programming

The idea of creating a software that is able to perform multiple operations simultaneously came together with computer science. In fact, the first concurrent code can be found in the Dijkstra article *"Solution of a Problem in Concurrent Programming Control"* [1] of 1965, just three years after the establishment of the first academic course in computer science in the United States.

"A concurrent system is one in which multiple tasks can be in progress at any instant" [2], thus time-sharing is the first attempt of concurrent computing, that allows a single processor to create the illusion of executing more applications simultaneously. This is reached by running a program for a time-slice and then allowing another process to gain the control of the CPU for the succeeding slice. Multitasking Operating Systems and Database Management Systems are well-known examples of concurrent systems on which a multitude of application are executed concurrently, even if only one processor is available.

Thanks to time-sharing, developers can write applications in which a multitude of tasks is specified and to run them on a single processor. Clearly, the capability of separating different execution flows in the source code simplifies the development of complex applications, giving an improved maintainability. On the other hand, the presence of multiple concurrent tasks might require synchronization when accessing shared resources in order to manipulate them correctly. In fact we cannot assume that tasks are executed in a deterministic order and are able to complete the manipulation of a shared resource in a known number of time-slices. Thus we have to guarantee

mutual exclusion [1] during the access of a shared resource by allowing only one task to execute the critical section, namely, a portion of code that needs to be executed in isolation by one task at a time. The most intuitive solution to guarantee mutual exclusion is the one of suspending tasks that want to access a critical section executed by another task. Anyhow, in a single processor machine, making some tasks to wait means giving an order of execution on them, without affecting the CPU utilization.

Since the eighties, for twenty-five years the speed of processors doubled every year. This led developers and users to believe that their programs could be faster by using a new generation of processors, regardless of whether the application is sequential or concurrent. The continuous upgrade of processor performance was predicted by Moore's Law [3], i.e., the observation that the number of transistor in an integrated circuit increases exponentially in time. Increasing density of transistors and their resulting smaller size allow to increase the speed of processors, giving improvements to any program. As stated in [4], since the 2000s the Moore's Law changes its manifestation. In particular the processors do not increase their sequential performance, but their number of cores, becoming ever more parallel. The reasons behind this trend reversal are a set of *"walls"*, i.e., issues in the current technology, that represent insurmountable obstacles to performance improvement of single core processors.

The power required by a core increases as the clock frequency, but a part of it is consumed in heat. In order to maintain an acceptable level of reliability, extreme high speed processors have to be cooled with very sophisticated solution, that are too expensive for commodity hardware. Thus, the effort to reach higher frequencies is unprofitable, leading to the *Power Wall*.

In order to increase single core performance, manufacturers make processors capable to perform more operations at a time by instruction level parallelism (ILP). For example, some processors can manipulate more data in a single instruction (Very Long Instruction), but its usage can limit the portability of the application. Another strategy consists in increasing the number of pipeline stages allowing to process more instructions at the same time. Since each stage of the pipeline executes different parts of different instructions, in case of conditional branches, we have to guess the result of

the next instruction. If we are lucky, the program can continue at full speed. In case of misprediction, we have to flush the whole pipeline, because we are executing wrong instructions. If we have a very long pipeline, we have to guess more frequently and the probability of flush increases, leading to more wasted time and worse performances. The most recent approach to achieve ILP is superscalar execution that consists in replicating some execution units (e.g. ALU) in order to process multiple instructions in parallel. In [5] the author explains that making ILP speedup significant requires very strong assumptions such as perfect caching and unlimited replicated resources. Removing them vanishes the payoff of extreme ILP. Moreover those solution have a super-linear complexity and power consumption, but they do not guarantee a linear speedup, giving the *Instruction Level Parallelism Wall*.

The *memory wall* is the last obstacle to performance improvement of processors. The speed of memory accesses increases exponentially, but in a smaller measure than the microprocessor speed improvement. As observed in [6], the gap between two exponential functions follows an exponential, leading main memory to be the major bottleneck in current and near-future applications.

Having an increased computational power means that a known problem could be resolved faster or with more accuracy, and, at the same time, a new set of problems can be faced. In other words, we can always find a way to exploit new computational resources and the demand for higher computational power has a perpetual growth. Due to the *power* and *ILP walls*, manufacturers tend to use available density of transistor in order to satisfy the increasing request of computational power by increasing the number of cores in a CPU. Clearly this trend has its limit in the *memory wall*. In fact more and more cores will fight for accessing same main memory locations and cache line. It is reasonable to think that the speed of air-cooled processors has reached its limit, but in future the number of cores per processor is destined to increase. This means that developers have to deal with parallel programming in order to exploit the multicore architecture of current and near future processors.

Parallel programming is a kind of concurrent programming. The difference between them is that, in parallel programming, we deal with hardware that allows to execute more tasks at the same time instant and this parallelism is visible to the developer.

As in time-sharing programming, in a parallel program several tasks cooperate in order to reach a common goal, typically sharing a set of resources. Thus, they need to coordinate through some form of communication and synchronization, making parallel programming harder than sequential one. The idea is trying to partition the computation and to distribute it to available processing units. The most used approaches are *task-parallelism* and *data-parallelism*. In the first case, we have to distinguish independent sets of tasks and then execute them in parallel on different cores. The second one consists in partitioning the working set in smaller parts that can be processed independently and in parallel. Both strategies are not a definitive solution. Task-parallelism assumes that we have enough tasks to make use of available cores. On the contrary, data-parallelism cannot be applied to any program. This implies that, in most cases, we have a combination of them in order to overcome their individual limitations.

Since several operations are executed at the same time, synchronization has new consequences when compared to time-sharing. In particular, we have explained that mutual exclusion allows one process at a time to access the critical section, while others have to wait. This creates a serialization that, considering a concurrent execution on a multicore machine, makes the application effectively use one core at time. In this case, no matter how many processors are available, our application is wasting a significant amount of resources and, at the same time, it is not able to take advantage of the addition of new processors.

This explains why an application does not run two times faster when executed on two processors, four times faster on four processors and so on. *Amdahl's Law* [7] poses a theoretical limit to maximum speedup achievable by parallelism, observing that any program has a sequential section which cannot be parallelized due to synchronized accesses to shared resources. If f is the percentage of code that can run in parallel on p processors, the speedup is:

$$Speedup(f, p) = \frac{1}{(1 - f) + \frac{f}{p}} \tag{1.1}$$

It means that if 90% of the program is parallelizable and we have an unlimited number of processors, the maximum speedup is:

$$\lim_{p \to \infty} Speedup(f, p) = \frac{1}{1 - 0.9} = 10$$

The program cannot run more than 10 time faster, regardless of the number of processors. Thus, speedup is bounded by critical sections. This is the reason behind the raising up of *non-blocking* algorithms, that limit synchronization to the execution of an atomic instruction provided by the underlying hardware architecture.

1.1 Parallel Hardware

With parallel hardware we mean a hardware that is able to perform multiple operations at the same time and this parallelism could be visible to programmers. In fact, even if a processor can handle multiple instructions at a time by using pipelining and superscalar hardware, these techniques are transparent to developers, who has to carefully design its program in order to exploit their potential, because it is not possible to control them directly. On the contrary, in parallel programs we have to explicitly deal with multiple entities (processes or threads), that can be executed at the same time on several processors. Processors share a medium that is used to communicate and coordinate the computation among them. This medium can be shared memory, a network or both. According to the used communication medium, we can distinguish between parallel and distributed programming. Parallel programs use shared memory to communicate, while in distributed programming processors communicate by sending messages through a network. Other authors make a different distinction between parallel and distributed programming [8]:

- *"in parallel computing, a program is one in which multiple tasks cooperate closely to solve a problem;"*
- *"in distributed computing, a program may need to cooperate with other programs to solve a problem."*

According to Flynn's taxonomy [9], we can distinguish several kinds of parallel

hardware depending on the number of data streams and instruction streams that it can handle at the same time. Thus a single-core processor is a Single Instruction stream and Single Data stream (SISD) machine. In fact instructions of every program are serialized building one flow of instructions, which manipulate a single data (excluding vector instructions). This flow is then executed by processors using techniques, such as pipelining or superscalar processing, that allow them to run multiple instructions of one flow at a time. In order to have a parallel machine, we need to increase the number of instruction streams, data streams or both.

Multiple Instruction streams, Single Data stream MISD allows to perform multiple instruction on the same data. We do not know a hardware machine that is universally recognized as MISD.

Single Instruction stream, Multiple Data streams A SIMD hardware is a synchronous system, because every computational unit executes the same instruction on different data at the same time. Thus the developer can exploit data-partitioning by writing a single sequential program that is executed in parallel for each unit of data. This allows to reduce the complexity of controlling units and to increase the number of ALUs in the chip, exploiting data parallelism. Graphic Processing Units (GPUs) fit into this category.

Multiple Instruction streams, Multiple Data streams The last category is MIMD, that can be modeled as a collection of Single Instruction stream processors connected through a communication medium, which could be a network or a firmware interconnection of processors. Thus a MIMD system is usually asynchronous because each processor executes its own stream of instructions and, even if they are running the same program, we cannot assume that they are executing the same instruction at a given instant. As said before processors are connected with a network and to a memory. According to the mutual logical position between memory and communication network we can distinguish two kinds of systems:

Shared Memory Systems Processors are connected first to the interconnection medium and then to memory, thus, they can access to any memory location.

Consequently communication among processors is made implicitly by writing and reading memory locations.

Distributed Memory systems Each processor is directly connected to its own private memory and can reach other processors through the network. A processor, that wants to access a location of the private memory of another processor, has to explicitly ask for it by sending a message.

1.2 Shared Memory

In shared memory systems, every memory location is accessible by each processor and communication among cores can be achieved by writing and reading memory locations. Shared memory is proper of multicore processors and, according to the core view of memory, we can distinguish:

Uniform Memory Access Each core is directly connected to the whole memory, thus it can access any location with the same latency and speed.

Non Uniform Memory Access Each core has some memory locations that are accessed faster than others. A core is directly connected to a block of memory (close memory, fast access), while it delegates accesses to other blocks to a special hardware embedded in the processor (far memory, slow access). Each block of memory is called NUMA node and more core can be connected to the same NUMA node.

1.2.1 Synchronization Primitives

Synchronization primitives are a set of atomic instructions offered by hardware. An atomic instruction allows us to execute a set of operations that appears to the system as performed instantaneously at an indivisible instant of time. Generally shared memory systems offer atomic read and write from/to memory of 1,2,4 bytes until accessed memory is aligned to 1,2,4 bytes respectively. On 64-bits systems read and write of 8 bytes are guaranteed to be atomic if aligned to 8 bytes. Other atomic synchronization primitives are:

`test&set`(mem, i) returns false if the i-th bit of the mem location is already set, otherwise sets the bit and return true;

`fetch&add`(mem, val) increments the value contained in memory location mem by val and writes the computed value into mem;

`compare&swap`(mem, old, new) returns false if the value contained in mem is not equal to old, else writes new into mem and return true.

In [10], the author explains that each synchronization primitive can resolve a specific class of the *consensus* problem. In particular the `compare&swap` (denoted as `CAS`) is a universal primitive, since it can resolve the consensus problem with any number of processes.

A problem associated with the usage of `CAS` is the ABA problem. This comes from the fact that a `CAS` cannot distinguish if a given value is written multiple times into a memory location. In fact, suppose that a thread T reads a value A from a shared memory location m and then another thread T' updates twice the value of m, first from A to B and then back to A. At this point, T performs a successful `CAS`. If T presumes that m has been untouched, it can leave the shared object, that contains m location, in an inconsistent state.

A possible solution for ABA problem is adding to the value a signature of the update, reducing the number of usable bits for representing the data. Conversely, if the value represents a memory reference, it means that the ABA problem occurs due to memory re-usage. Under the assumption that a shared variable cannot assume twice a given address until the referenced memory is released and allocated, a garbage collector system (see Section 1.6) can resolve the issue. In fact, it denies the capability of freeing memory (and thus reusing it) until any thread holds a reference to it.

1.3 Progress Conditions

As said above, in a concurrent program a set of tasks cooperate to reach a common goal, sharing a set of resources. With *process* or *thread* we mean the execution of an instruction stream. Concurrent accesses to a shared resource not being serialized

might produce unpredictable results, called *race conditions* [11], that depend on the non-deterministic interleaving of tasks' executions. A section of a program that can lead to a race condition is a critical section. For our purpose we are interested in considering the shared resource as a memory object, i.e., data that can be manipulated by using a set of procedures called *primitives*. The definition of an object is unrelated to the application concurrency, thus the preconditions and postconditions of primitives have sequential definition. It implies that accessing a shared object leads to a critical section, which should be protected in order to guarantee mutual exclusion. This goal could be reached by delaying a process that wants to enter in a critical section executed by another process. However, if the process in the critical section is suspended, those that are waiting cannot resume the execution. This leads to an underutilization of the computational resources because no one is proceeding. We define as *blocking* a procedure that has a critical section and a process executing it can be suspended otherwise it is a *non-blocking* procedure. A memory object is non-blocking if all its primitives are non-blocking. From this definition it follows that, in a non-blocking algorithm, the critical section must be an atomic instruction, because it ensures that a process cannot be suspended during its execution.

A progress condition is a procedure property that guarantees some level of evolution in the system state. We can distinguish several kinds of progress depending on whether the procedure is blocking or non-blocking [12]. In the first case, the weakest progress condition is *deadlock-freedom*. A procedure is *deadlock-free* if it is guaranteed that some thread eventually executes its critical sections. This condition guarantees that system makes progress, but does not ensure the same property for each thread. On the contrary, a *starvation-free* procedure ensures that every thread eventually executes the critical section, thus making progress. The progress conditions for blocking algorithms are *dependent* from the scheduler, because they need that each thread performs infinite operations in an infinite period and, consequently, it can complete a critical section. Non-blocking algorithms have specular progress condition and are called *independent* because they can perform any number of operations in an infinite time. In particular *lock-freedom* guarantees that some method eventually returns and *wait-free* ensures that every method eventually returns. Another progress condition that is proper of

non-blocking procedures and strictly weaker than lock-freedom, is *obstruction-freedom*, which guarantees that a procedure returns if executed in isolation, i.e., in a period without contention with other threads on the same data.

In general a programmer assumes that the operating system is benevolent and has a scheduler which ensures each thread to advance. This assumption promotes lock-freedom and deadlock-freedom to their respective upper-level progress conditions, wait-freedom and starvation-freedom respectively.

1.4 Correctness Conditions

Correctness conditions allow to understand the properties that concurrent objects have to satisfy in order to develop a relevant result. In particular dealing with concurrency means developing in an environment that is intrinsically non-deterministic, because no assumption can be made on the interleaving among operations performed by different processes. More feasible is understanding if a sequential execution is correct or not. Obviously, a valid sequential execution is one compliant with the object definition. Objects have a sequential definition, but they live in a concurrent environment, thus, in order to demonstrate the correctness of a concurrent execution, we have to find an equivalence with a sequential one and check if it is valid. Before analyzing the correctness conditions, we give the notion of *history*.

A history models a process execution and it is a finite sequence of *call* and *return* events [13]. Clearly this a simplification of a process execution, that can involve a huge number of read and write on several shared memory locations, but we can model those operations as a pair of call and return. This means that the given definition of history is not restrictive to memory objects, but can be applied to any complex system that accesses memory. A history is *sequential* if: every call in the history, is immediately followed by its response; each response is immediately preceded by its call. A *concurrent* history is a history that is not sequential. Given a process and a history, the *process subhistory* is the subsequence of all events performed by the considered process in the history. Given that a process is a sequential execution, in a history all process subhistories are sequential histories. In this case the concurrent

history is *well-formed*. A *object-history* is a subhistory in which all events are related to a given object, thus it is not necessarily a sequential history. A sequential history is legal if each object-history is correct according to the sequential definition of the object itself. The given definition of history can capture the concurrency generated by any kind of parallelism (time-sharing, hardware parallelism, etc.), because it relies only on call and return events. Now we can define the equivalence between two histories. Two histories are equivalent if for every process, the process subhistories are identical in both histories. Finally, in order to evaluate the correctness of a concurrent execution, we have to check its equivalence with a chosen sequential one, that satisfies a given set of properties.

We report three conditions: *sequential-consistency* [14], *serializability* [15] and *linearizability* [13] and we explain their implication by using as example some executions that manipulate a FIFO queue. A FIFO queue is a data structure that consists in a set of items on which two procedures are defined: *enqueue* that adds a given element to the set; *dequeue* which removes from the set and returns the element inserted before than others. In the examples, the signature $\texttt{op}_i(x)$ indicates a procedure *op* invoked by a thread i with parameter x, while $\texttt{ret}_i(v)$ means that the last procedure of the thread i has returned the value v (since we are considering only *well-formed* history).

1.4.1 Sequential Consistency

Sequential consistency, defined by Lamport [14] requires that a history h has to be equivalent to a valid sequential history h'. It means that the routines invoked by a thread have to appear in the same order in h', while it is not required to maintain the order among operations of different threads. In the example, the history H is sequentially consistent because it is equivalent to the sequential history H', even if this one does not follow real-time order.

1.4.2 Linearizability

A history is linearizable [13] if it is equivalent to a sequential history such that any couple return-call appears in the same order in both histories. Linearizability is similar

H:
1. $\texttt{push}_A(x)$
2. $\texttt{ret}_A()$
3. $\texttt{push}_B(y)$
4. $\texttt{ret}_B()$
5. $\texttt{pop}_B()$
6. $\texttt{ret}_B(y)$

H':
1. $\texttt{push}_B(y)$
2. $\texttt{ret}_B()$
3. $\texttt{pop}_B()$
4. $\texttt{ret}_B(y)$
5. $\texttt{push}_A(x)$
6. $\texttt{ret}_A()$

to sequential consistency, but imposes that the reference sequential history maintains real-time ordering among completed procedures. Conversely concurrent invocations can have an arbitrary order among them as long as the resulting history is valid. Informally it is like concurrent procedures appear as executed in an atomic point between the invocation and the return and the order among those points is compatible with the sequential definition of the object. The concurrent history H'' is equivalent to H''', thus is linearizable, while the history H of the previous paragraph is not linearizable since no reordering is possible (it is a sequential history).

H'':
1. $\texttt{push}_A(x)$
3. $\texttt{push}_B(y)$
2. $\texttt{ret}_A()$
4. $\texttt{ret}_B()$
5. $\texttt{pop}_B()$
6. $\texttt{ret}_B(y)$

H''':
1. $\texttt{push}_B(y)$
2. $\texttt{ret}_B()$
5. $\texttt{push}_A(x)$
6. $\texttt{ret}_A()$
3. $\texttt{pop}_B()$
4. $\texttt{ret}_B(y)$

1.4.3 Serializability

Serializability [15] imposes that a history is equivalent to a valid sequential one, in which event of different process subhistory cannot interleave. This definition was designed for Data Base Management Systems in which a process is a *transaction*, i.e. a set of ordered operations on different objects that has to appear to the system as executed in atomicity and isolation. *Strict-serializability* maintains the real-time order among transactions. It means that if the transactions involve one operation on a single object serializability collapses into the sequential-consistency and strict-serializability into the linearizability.

1.5 Non-Blocking Data Structures

In this section we analyze some lock-free data structures that are useful in order to understand non-blocking synchronization. For each proposed example, we will explain implementation details and give some sketch proof of lock-freedom and linearizability. The algorithms analyzed use `CAS` as synchronization primitive, since it is universal and, as stated in [12], it is both necessary and sufficient to reach lock-free synchronization when paired with atomic read and write. In the proposed algorithms there is not any explicit handling of the ABA problem in order to maintain the focus on the strategy used to obtain lock-freedom. Moreover, we believe that algorithms built on top of `CAS` primitive should ignore ABA problem at first, because we can build a software `CAS` on top of this hardware primitive, which guarantees a higher level of robustness. Similar assumptions are made on two other aspects. The first one is the implementation of the memory allocator. In fact if we want to use a lock-free object, we have to guarantee lock-freedom for routines, used by the object, that allocate and free memory. A simple trick, that allows to keep our object lock-free when using a blocking allocator, is moving every allocation before each invocation of object routines and every release after. Unfortunately, in order to apply this strategy, we have to know in advance how many and which objects are allocated in procedure execution, but this is not always possible. Anyhow, if we are interested in using a non-blocking object, probably we are building an algorithm that we would like to be non-blocking, thus the problem still holds. Consequently we have to rely on non-blocking allocators. Finally in a non-blocking algorithm we need a garbage collection mechanism in order to safely release memory, but the reason behind this is explained in the next section.

System Model The reference system is a MIMD multi-processor with shared memory, in which a program specifies a set of cooperating tasks, each one executed sequentially in a processor. The execution of a task is called *thread* or *process*. The synchronization primitives available are atomic read,write (non explicit in the proposed algorithms) and atomic `compare&swap` denoted as `CAS`. As the correctness criterion we adopt the linearizability.

```
TreiberStack{
  pointer<TreiberStack> top;
  Value key;
}
```

Figure 1.1: Treiber's Stack object.

(a) Push (b) Pop

Figure 1.2: Visual representation of Treiber's Stack procedures.

1.5.1 Treiber's Stack

The Treiber's Stack [16] is the first example in literature of a non-blocking algorithm. A stack is a collection of elements that implements a Last-In First-Out policy. Two primitives are defined, *push* and *pop*. In particular *push* routine add an element to the stack and *pop* returns the last element added with a *push*. The stack consists in a pointer to a node that represents the top (Figure 1.1). Each node contains a value and a pointer to the successive element in the stack. *push* routine encapsulates the value in a new node, which points to the current head of the stack, and then tries to swap it with the new node using an atomic `CAS` (Figure 1.2(a)). The exchange is retried until it succeeds. The *pop* procedure has to remove the current top. This is obtained making the variable top to point its successor atomically with a `CAS` (Figure 1.2(b)) and repeating the operation in case of failure. In Figure 1.3 is shown the complete algorithm.

Treiber's Stack is a lock-free data-structure, because it guarantees that at least *some* thread make progress. In fact if a procedure is blocked in a repeat-until loop, it means that every attempt of performing a `CAS` fails. It follows that for each failed `CAS` there is a successful one performed by another thread. The algorithm is not wait-free because a procedure might perform an infinite number of attempts, thus without returning. The

```
TreiberStack.push(Value v){
 1: new ←TreiberStack()
 2: new.key ← v
 3: repeat
 4:     new.top ← top
 5: until CAS(&top, new.top, new)
}
```

```
Value TreiberStack.pop(){
 1: repeat
 2:     tmp ← top
 3:     new ← tmp.top
 4: until CAS(&top, tmp, new) return
    tmp.key
}
```

Figure 1.3: Treiber's Stack algorithm.

```
HarrisSet{
  pointer<Node> head;
  pointer<Node> tail;

  HarrisSet HarrisSet()
   1: head←Node()
   2: tail←Node()
   3: head.next←tail
}
```

```
Node{
  pointer<Node> next;
  Value key;

  Node Node(Value v)
   1: new ←Node()
   2: new.key ← v  return new
}
```

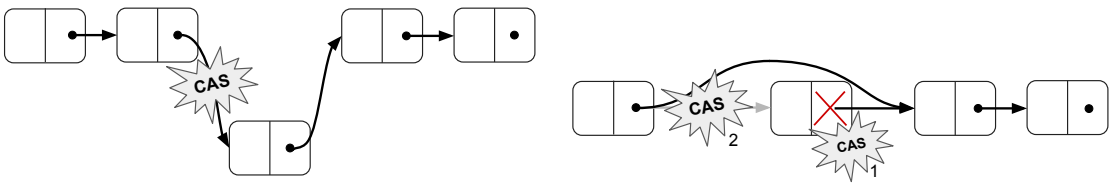
Figure 1.4: Harris' Sorted Linked List object.

algorithm is also linearizable. As an informal proof, we can observe that if an history is incomplete, it means that some procedure is stuck in the try-loop. Their execution cannot modify the stack, so we can safely remove them from the history obtaining a complete history. For each complete history we can always find a linearization point in the CAS execution. In other words procedures appear to be executed atomically.

From scalability point of view, Treiber's Stack has a sequential bottleneck in the top field. In fact each thread competes to update the same field and at the same time they cannot take advantage from cache usage, because the most access field is constantly updated, leading to lower performances than locking alternatives. In [17], the authors resolve this bottleneck introducing a backoff mechanism based on a front-end array that allows to reduce collisions on the top variable.

1.5.2 Harris' Sorted Linked List

A Linked List consists in a chain of nodes that contain an object. Each node stores a pointer to the next node. Harris' Sorted Linked List [18] implements the semantics



(a) Insert (b) Delete

Figure 1.5: Visual representation of Harris' Sorted Linked List procedures.

of a set of keys on which exist a total order. Thus three procedure are defined: *insert* add the key if it is not present in the set; *delete* remove a given key from the set; *find* check if the set contains a given key. An empty set contains two sentinel nodes, head and tail, and every node containing a key is inserted between them (Figure 1.4). Procedures use a private routine *search*, that, given a key, it finds two valid adjacent nodes such that the left node has a key strictly lower than the search key and the right node has key greater than or equal to the search key.

The *insert* routine uses *search* in order to find left and right nodes of the search key. After checking that the right node does not contain the key, the algorithm tries to insert a new node between left and right ones by using a `CAS` on the next field of the left node (Figure 1.5(a)). *find* simply calls *search* routine and checks if the right node has a key equal to the search one. *delete* searches for a right node with the desired key and, if it is present, it tries to logically remove it by marking the next field with a `CAS`. The mark is stored on the least significant bit of the next field, which is a reference. Thus, assuming a 2 bytes alignment (on 32-bit machines it is generally 4 bytes), we have guaranteed that each reference has an even address value and consequently the default value of the first bit is always zero. A marked node is a node such that its next field has the first bit equal to one. Logically deleted nodes are physically removed during a *search* execution. This routine finds two unmarked nodes that might be not adjacent, because there are some marked nodes between them. In this case the algorithm tries to make them adjacent by a `CAS` (Figure 1.5(b)).

Lock-freedom is guaranteed by the fact that the `CAS`es disconnecting marked nodes can be executed a number of times which is bounded by the number of logically removed nodes. It means that eventually only `CAS`es for marking and inserting a node will be

```
HarrisSet.find(Value v)
 1: ⟨ _ , rightNode⟩ ← search(v) return
    rightNode ≠ tail ∧ rightNode.key = v

boolean HarrisSet.insert(Value v)
 1: new ← Node(v)
 2: repeat
 3:     ⟨leftNode, rightNode⟩ ← search(v)
 4:     if
    rightNode ≠ tail ∧ rightNode.key = v
    then return false
 5:     end if
 6:     new.next ← rightNode
 7: until CAS(&leftNode.next, rightNode,
    new) return true

HarrisSet.delete(Value v)
 1: repeat
 2:     ⟨leftNode, rightNode⟩ ← search(v)
 3:     if
    rightNode ≠ tail ∧ rightNode.key ≠ v
    then return false
 4:     end if
 5:     tmpNext ← rightNode.next
 6: until ¬ISMARKED(tmpNext) ∧
    CAS(&rightNode.next, tmpNext,
    MARK(rightNode))
 7: if ¬CAS(&leftNode.next, rightNode,
    tmpNode) then
 8:     search(v)
 9: end ifreturn true

⟨Node,Node⟩ HarrisSet.search(Value v)
S1: tmp ← head
S2: tmpNext ← head.next
S3: repeat
S4:     if ¬ISMARKED(tmpNext) then
S5:         left ← head
S6:         leftNext ← head.next
S7:     end if
S8:     tmp ← UNMARK(tmpNext)
S9:     if tmp = tail then
S10:        break
S11:    end if
S12:    tmpNext ← tmp.next
S13: until ¬ISMARKED(tmpNext) ∧
    tmp.key ≥ v
S14: right ← tmp
S15: if leftNext = right then
S16:    if right ≠ tail ∧
    ISMARKED(right.next) then
S17:        goto 1
S18:    elsereturn ⟨left, right⟩
S19:    end if
S20: end if
S21: if CAS(&left.next, leftNext, right) then
S22:    if right ≠ tail ∧
    ISMARKED(right.next) then
S23:        goto 1
S24:    elsereturn ⟨left, right⟩
S25:    end if
S26: end if
```

Figure 1.6: Harris' Sorted Linked List algorithm.

executed. Supposing that each operation cannot terminate, this is possible if and only if each CAS contained in each operation fails. Each operation when successful has at most two successful CAS, one in the search execution and one for insert or delete. If some thread cannot terminate it means that every CAS fails and consequently some other thread executes a successful CASes. If those CASes are executed only in search routine, no thread makes progress, but this is impossible because the number of marked node is bounded by the successful delete operations. On the contrary if the number of marked node increases it means that other threads are able to insert and to delete

nodes, thus they have progress.

The Harris' Sorted Linked List is a linearizable data-structure. As linearization points we can use every successful `CAS` that inserts or marks a node. Finally considering instants in which post-conditions of every successful search are satisfied, they impose a real-time order for *find* invocations.

Under high contention, the linked list has a more scalable behavior than Treiber's Stack, because conflicts are more likely spread on all nodes in the list allowing a greater number of operations to be successful at the same time. The complete algorithm is shown in (Figure 1.6).

1.5.3 Lock-Free Dynamic Vector

A vector is a memory object that allows to maintain a collection of elements and to access them by using an index. The capacity of a vector can be explicitly reserved or dynamically increased and reduced when a new element is respectively added and removed at the end of the array. Thus a vector specifies: *push_back* adds an element at the end; *pop_back* deletes the last element; *read_at* and *write_at* access the element at a given position; *reserve* which preallocates memory in order to guarantee that the vector size is sufficient to contain a given number of elements; *size* that allows to query the actual size of the vector.

In [19], the authors develop a lock-free implementation of a vector data type. In this work every update that needs to modify more than one memory location is performed through the usage of a *descriptor*, which contains every useful data to complete its operation. In particular both *push_back* and *pop_back* can manipulate the last element of the vector, increase or decrease the vector size and allocate new space for new entries. Thus, they are descriptor-modifying operations. When a process tries to perform a descriptor-modifying operation, it tries to exchange the current descriptor by `CAS` with a new one. Before publishing its own descriptor, it checks if there is a pending operation by analyzing the current descriptor and, if required, tries to complete it. Non descriptor-modifying operations are served in a wait-free fashion. In particular *read_at* and *write_at* perform a direct access to the vector element, while *size* retrieves the current size from the current descriptor and returns it, decreasing the value if there

```
Vector{
  pointer<Descriptor> descriptor;
  pointer<Value> memory[32];
  integer init_size;
  integer fbit;

   Vector Vector(integer start_up_size)
    1: new ←Vector()
    2: new.init_size ← start_up_size
    3: new.fbit ← ibsr_x86(start_up_size)  return new
}
```

```
Descriptor{
  pointer<WriteDescriptor> pending;
  integer size;

   Descriptor Descriptor(integer
          size, WriteDescriptor write_des)
    1: new ←Descriptor()
    2: new.size ← size
    3: new.write_descriptor ←
    4:                write_des  return new
}
```

```
WriteDescriptor{
  Value old;
  Value new;
  integer pos;
  boolean pending;

   WriteDescriptor WriteDescriptor
              (Value o, Value n, integer p)
    1: new ←WriteDescriptor()
    2: new.old_value ← o
    3: new.new_value ← n
    4: new.pos ← p
    5: new.pending ← true return new
}
```

Figure 1.7: Lock-Free Dynamic Vector List object.

is a pending write. The *reserve* routine checks that the current allocated space is sufficient to contain the given number of elements and eventually it allocates a new block, thus it is a non descriptor-modifying operation. The definition of descriptor and vector are shown in Figure 1.7.

The most critical operation in a non-blocking array is the physical expansion. The simplest solution consists in copying the array and exchanging its reference with a CAS, but its hardly linearizable because already copied elements can be updated and some process can still hold an old pointer to the array. It means that each cell of the array has to contain a pointer to the value, thus we can copy and exchange the array safely. This leads to increase space overhead by a factor, that can be significantly large for primitive types (at least 2). Moreover accessing an element of the array requires an additive memory access, because we have to read first the pointer to the value and then we can access the value. To resolve this issue, the authors propose a vector

implementation that never copies allocated memory, by using a two-level indexing. The first level is a fixed size array such that each entry contains a pointer to a contiguous block of memory which stores elements. When a resize is required, the algorithm allocates a new memory block with size equal to the current one and its reference is stored into the next free entry of the first level array. Finally the two-level indeces are mapped to linear indeces in order to access the vector as if it were a contiguous memory area. The mapping is computed efficiently by making the initial size of the array a power of two and by doubling it at every expansion. In particular, let D be the initial size of the array, thus it grows as $2^0D, 2^1D, \ldots, 2^nD$. After each expansion a new memory block wide as the previous size is allocated. For example after the 1-th expansion a new block of size D is added, during the 2-th one a $2D$ wide block is allocated and so on. Each new block is stored sequentially in first level of the bi-dimensional array. It means that the i-th element of the first array points to a 2^iD wide array and covers the linear indeces range $[2^iD, 2^{i+1}D)$, or equivalently $[2^{i+d}, 2^{i+d+1})$, where $d = \log_2(D)$. An index $I \in [2^{i+d}, 2^{i+d+1})$ can be written as $I = 2^{i+d} + R$, thus it is mapped to the pair $\langle i, R\rangle$, where $i = \lfloor \log_2 I \rfloor - \lfloor \log_2 D \rfloor$ and $R = I - 2^{\lfloor \log_2 I \rfloor}$ and. This allows to take advantage from Bit Scan Reverse instruction that returns the index of the more significant bit equal to 1, computing the discrete logarithm in base 2. The complete algorithm is shown in Figure 1.8.

As explained above, non descriptor modifying operations are wait-free, because they consist in one atomic operation (read, write or `CAS`), guaranteeing that each one always completes. On the contrary, descriptor-modifying procedures are lock-free because descriptor update is performed in a try-loop on the result of a `CAS`.

In order to keep the data-structure linearizable and efficient at the same time, the authors impose that non descriptor-modifying operations on the bottom of the vector have to be never executed concurrently with a *pop_back* routine. If we consider a *pop_back* that completes before a concurrent read or write at the last element, we have a non linearizable execution. The authors suggest two strategies for this problem. The first one consists in making each operation descriptor modifying, thus reducing efficiency and loosing wait-freedom for write and read routines. Alternatively they define the following usage rule: *"Non-descriptor modifying operations that access the*

```
Vector.pushBack(Value elem)
 1: repeat
 2:     curDes ← descriptor
 3:     tmpSize ← curDes.size
 4:     completeWrite( curDes.pending )
 5:     bucket ← ibsr_x86( tmpSize
 6:                                + init_size )
 7:     bucket ← bucket − fibit
 8:     if memory[bucket] = null then
 9:         allocBucket(bucket)
10:     end if
11:     tmpValue ← read( tmpSize )
12:     writeop ← WriteDescriptor
13:                 ( tmpValue, elem, tmpSize )
14:     newDes ←
15:             Descriptor(tmpSize + 1, writeop)
16: until
17:         CAS(&descriptor, curDes, newDes )
18: CompleteWrite(newDes.pending)

Value Vector.popBack()
 1: repeat
 2:     curDes ← descriptor
 3:     CompleteWrite(curDes.pending)
 4:     elem ← At(curDes.size−1)
 5:     newDes ←
 6:             Descriptor(curDes.size−1, null)
 7: until
 8:         CAS(&descriptor, curDes, newDes )
    return elem

integer Vector.size()
 1: des ← descriptor
 2: size ← des.size
 3: if des.writeop.pending then
 4:     size ← size − 1
 5: end ifreturn size

Value Vector.read(integer i)
        return atomic_read(At(i))

Vector.write(integer i, Value elem)
 1: atomic_write(At(i), elem)

Vector.reserve(integer size)
 1: i ← ibsr_x86(descriptor.size +
    init_size−1) − fibit
 2: if  i < 0 then
 3:     i ← 0
 4: end if
 5: while i <ibsr_x86(size + init_size−1) −
    fibit do
 6:     i ← i + 1
 7:     allocBucket(i)
 8: end while

Value Vector.at(integer i)
 1: pos ← i + init_size
 2: hibit ← ibsr_x86(pos)
 3: idx ← pos xor 2^hibit return &memory[hibit
    − fibit][idx]

Vector.completeWrite(
                    WriteOperation writeop)
 1: if writeop.pending then
 2:     CAS(At(writeop.pos), writeop.value_old
    , writeop.value_new )
 3:     writeop.pending ← false
 4: end if

Vector.allocBucket(integer bucket)
 1: mem ← new Value[init_size^(bucket+1)]
 2: if ¬CAS(&memory[bucket], null, mem) then
 3:     free(mem)
 4: end if
```

Figure 1.8: Lock-Free Dynamic Vector List algorithm.

tail should never be executed concurrently with descriptor modifying operations that reduce the vector's size", but without suggesting how to satisfy it. It follows that read and write operations do not provide bound checking. As a proof sketch of linearizability we observe that for every operation it is possible to find a linearization point such that it seems executed atomically and instantaneously. For non descriptor-modifying operations the linearization point is the instant in which the atomic read or write is

executed. On the contrary, descriptor modifying routines exchange the descriptor with a `CAS`. When successful, it is a linearization point for *pop_back*, while *push_back* takes its effect when the `CAS` of the corresponding write is performed.

1.6 Memory Reclamation For Non-Blocking Dynamic Objects

In previous section we have mentioned the fact that we need a garbage collector when implementing a dynamic non-blocking object. We try to make clear this need by using an example execution on a Treiber's Stack. Considering a *pop* procedure, if a thread succeeds to perform the `CAS`, then disconnected node can no longer be accessed by other threads. However this does not allow us to release the node. In fact, we have guaranteed that no thread can read a reference to the node after the successful `CAS`, but some thread can still hold it from a previous access. For example, a *pop* reads first the reference to the current top, then it reads the top field and finally tries a `CAS`. If after the first read, another thread disconnects and frees the node, the second read leads to dereference a dangling pointer i.e., a pointer to memory that is reused or no longer allocated. Similar patterns can be found in Harris' Sorted Linked List and Lock-Free Dynamic Vector. In general we can assume that procedures of a dynamic data structure manipulate a set of smaller objects that are interconnected to each other. Routines can allocate, release, connect and disconnect objects. The implementation of a data structure defines when and how objects have to be allocated, connected and disconnected. Allocation and release operations are implemented by a memory allocator. Finally a garbage collector is responsible to detect when objects can be released safely. Moreover, using a garbage collection mechanism allows to reuse memory without worrying about ABA problem.

Some reclamation techniques consist in identifying a *grace period*, i.e., a time interval such that each object allocated before the interval can be released after that interval. The start and end instant of the interval are discovered by making all threads pass through a synchronization point. Quiescent State and Epoch Based Reclamation fall in this category, while Hazard Pointers and Reference Counting use a protocol before each usage of a reference that might point to a released memory. An extensive

comparison of these reclamation strategies is given in [20], where there is not a clear winner.

1.6.1 Quiescent State Based Reclamation

A *Quiescent State* [21] is an instant in which a thread does not hold any reference to shared objects. Thus a time interval that contains at least one quiescent state of each thread is a grace period. A simple method to implement this is using an atomic counter and a flag. When a thread enters in a quiescent state it tries to set the flag in order to publish the beginning of a grace period and then increments the counter if it has not already done. After the increment, it checks if the counter has reached the number of threads. If not the thread retries in the next quiescent state, otherwise it frees objects that it has disconnected and decreases the counter if it has not already done. If the counter reaches zero it tries to reset the flag. In order to use this reclamation strategy, a developer has to explicitly define quiescent states, by calling a routine of the garbage collector.

The Quiescent State Based Reclamation is a blocking algorithm. In fact if a process is halted by a fault, grace periods can no longer exist and no thread can release memory, leading the system to block on memory allocation eventually.

1.6.2 Epoch Based Reclamation

Epoch Based Reclamation [22] is similar to Quiescent State Based Reclamation, but it applies a different strategy to compute the *grace period*. In particular it applies the concept of *epoch*. We can distinguish local and global epochs. A new global epoch starts when all threads, that are executing a non-blocking procedure, are in the same local epoch equal to current global epoch. At a first level of approximation, a local epoch is the last global epoch observed by a thread after it has performed a given number of non-blocking procedures. Thus, a grace period is made of multiple global epochs because when a thread enters in a new epoch it can observe objects from the previous epoch. Due to the fact that a new epoch beginning depends on local epochs of threads which are executing non-blocking procedures, a grace period spans at most

three epochs. More precisely, if current global epoch is e, threads in non-blocking regions are in epoch e or $e-1$, thus they can observe pointers in e, $e-1$ and $e-2$ epoch. It means that a thread can safely release objects that were disconnected three epochs before the current epoch.

In order to use Epoch Based Reclamation, developers have to signal the starting and the ending of each non-blocking section in the code.

This algorithm is blocking, but it is more reliable than Quiescent State Based Reclamation. In fact, only threads halted in the execution of non-blocking procedures can block the release of memory, because global epoch cannot advance.

1.6.3 Hazard Pointers

The idea behind Hazard Pointers [23] is to publish every reference that is used or going to be used by a thread, in order to prevent a premature release. The publishing consists in inserting a reference in a data structure that is visible to all other threads. When a thread disconnects an object, it checks that references to the object are not contained in the set of hazard pointers of other threads. If it is not present, a thread can safely release the object, otherwise it saves in a *release-later* list. The safety is guaranteed by disconnection that makes the object unreachable. On the contrary, it is possible that a thread holds a reference, but it has not updated its hazard pointers. Thus before accessing an object, a thread has to set the hazard pointer and then to check if the object is still connected. In order to perform this operation efficiently (e.g. without traversing the whole data-structure), a thread has to publish every pointer that is useful to reach the object. Considering a stack, one hazard pointer is sufficient, while for linked list two hazard pointers are required: one for current node and one for its predecessor.

The present solution is wait-free, in fact each thread only checks hazard pointers with a scan and updates its own hazard pointers. Thus it guarantees that memory is always released except for those objects that are addressed by hazard pointers of halted threads.

Hazard pointers require that the developer is able to identify hazardous accesses to shared objects accessed without lock and to individuate the minimal number of

pointers required to protect them.

1.6.4 Reference Counting

Reference Counting is introduced in [24] and consists in incrementing a counter associated with a reference before using it. The counter is decreased each time the reference is not used anymore. It means that when an object is disconnected from the data structure the counter can only decrease, thus when it reaches zero the object can be released. Although it seems to be very simple and effective, if it is not implemented carefully, it can perform worse than other reclamation schema [23].

In order to use reference counting, the developer has to modify every access to shared objects in each non-blocking sections of code.

Chapter 2

Pending Event Set Problem

Event-driven programming is used in a very wide range of fields that span from video-games to military and medical applications. In particular, it is widely used for implementing simulators that allow scientists to have a clearer knowledge of a phenomenon and to test solutions for a given problem without implementing them. This is extremely important during the design of a technology or before taking a decision that could affect safety of humans, companies and nature.

In event-driven programming the computation is based on a model that describes only system's aspects of interest and the information needed to represent it with the desired accuracy. This information is compound of variables that describe the conditions of the system in a given instant of time and, taken together, define the state of the system. If the system is made of several sub-systems, the system state is the join of sub-systems states. The model also defines the *events* that emulate occurrences in time of given conditions or signals. Each event is associated to a function, that models the *reaction* to the event, which produces changes to the system state and might generate new events. Thus, the evolution of the system is determined by the occurrence of events. When the duration of events is impulsive, i.e., the time instant of its beginning is the same of its ending, system variables can change only at points in time at which events occur. Thus, between two consecutive events the system state is untouched. Focusing resources on processing events that update the system state allows event-driven programs to gain a natural responsiveness to asynchronous and unpredictable environments.

The execution of an event e_0 can create new events $e_1, e_2, \ldots, e_n$ that causally depend from the first one. Since the timestamp T_{e_0} associated with the event e_0 represent the time in which the event occurs, we have that $T_{e_1} \geq T_{e_0}$. This guarantees that executing an event cannot affect the past and the logical time can advance. With logical time we mean the time in which events happen in the program and it is opposed to wall-clock time, which is the time measured by humans that are waiting for computation results. When the execution of current event is terminated, generated events are enqueued in a data structure, called Pending Event Set. The next event to be executed is retrieved with a dequeue procedure, which returns the event with minimum timestamp in the set. Executing the minimum is fundamental because, if events can be executed out-of-order, a causality error can occur. In fact, if an event e_i can be executed before an event e_{i-1} and $T_{e_i} \geq T_{e_{i-1}}$, the final state obtained by the execution of e_{i-1} at logical time $T_{e_{i-1}}$ is dependent from something happened in future at time T_{e_i}.

Generally, the data structure is a priority queue, i.e., a set of items associated with a key. On the keys it is defined a total order that reflects the objective policy. The next item to be dequeued is the one associated with the minimum key in the queue. Manipulating the strategies to assign keys on items, we can obtain several scheduling policy. The First-Input-First-Output (FIFO) policy consists in returning first the oldest element. This can be achieved by assigning monotonically increasing keys to items during the insertion. On the contrary, when it is required to return first the last inserted element, we have a Last-Input-First-Output (LIFO) policy, that can be obtained by using monotonically decreasing keys. In event-driven programs, the objective policy is the "happened first" (denoted as $\rightarrow$) relationship that reflects the causal order on events. This is obtained using timestamps as priorities keys. The notion of causality is captured by timestamps, in fact $e \rightarrow e' \Rightarrow T_e \leq T'_e$. The converse does not hold, in fact it is possible that an event e generates more events e', e'' with same timestamp $T'_e = T''_e$, that are unrelated. For this reason many implementation of PES are based on a stable priority queue. A priority queue is stable if events with identical priority are dequeued according to the order in which they are enqueued.

The management of the pending event set has a crucial role in performance, in

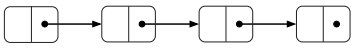


Figure 2.1: Visual representation of a linear object.

fact, for each event execution, multiple enqueues might be required in order to insert new events and one dequeue is needed to obtain the next event. The empirical study in [25] shows that PES management might reach the 40% of the overall execution of a Discrete Event Simulation, that is an instance of event-driven paradigm. This explains the extensive research in priority queue, that have produced an enormous number of different implementations. In following sections we introduce some examples of pending event set implementations, focusing on strategies used to arrange events.

2.1 Linear Pending Event Set

A priority queue can be implemented by using a linked-list (Figure 2.1) in which elements are stored in a chain that has to be traversed in order to reach a given position. If elements in a linked list are unsorted we have to scan the whole list for finding the minimum, while the enqueue can insert a new event at the beginning of the list. This leads to $O(n)$ steps for returning a minimum, where n is the number of items stored, and a constant cost for inserting a new event. On the contrary, if events are sorted according to the order defined on priorities space, a dequeue operation consists in removing the head of the list and returning it giving a constant cost, while the enqueue requires a scan in order to insert the new element at the correct position. In the worst case this requires that the whole list is traversed leading to a $O(n)$ cost. The median-pointer linked-list [26] is an ordered linked-list with the addition of a pointer to the node in the middle of the list. During an insertion, we can efficiently discover if a new node has to be inserted in the first half of the list or the second one, but this allows to skip $\frac{n}{2}$ events and the same amount of event has to be traversed. Thus the asymptotic cost is still $O(n)$.

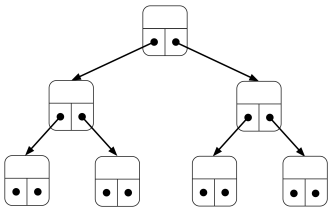


Figure 2.2: Visual representation of a tree-like object.

2.2 Tree Based Access Pending Event Set

A tree (Figure 2.2) is a data-structure based on nodes similar to the ones of the linked-list. The difference is that each node maintains references to multiple nodes called *children*. A node can appear only in the children list of one node, called *parent*. A *root* node is a node without parent. A *leaf* is a node without children. The *depth* of a node is the number of parents that has to be traversed from the root to the node. The *height* of a tree is the maximum depth. A *level* of a tree is the set of all nodes with a given depth. A *complete* tree is a tree in which leaves are on at most two levels and those are in the last level are as far left as possible. When a node can have at most two children, named *left* and *right* child, the tree is called binary. A binary tree can be efficiently stored in an array where each node at position k has its children stored at positions $2k$ and $2k+1$. In this case we have an *implicit* binary tree. Finally, a binary search tree is a binary tree in which, given a node, its left child stores a key lower than or equal to its key and the right child contains a key greater than its key.

The implicit binary heap [27] is an implicit, complete binary tree with the heap property, i.e., each children has a key greater than or equal to the parent key. The insertion consists in adding the new key in a new leaf, maintaining the completeness property. If heap property is violated, it is restored by exchanging the new node with its parent recursively. It means that the minimum is always stored in the root. Thus a dequeue consists in removing it and returning its value. Removing the root means exchanging its key with the key of the rightmost leaf, removing that leaf and restoring the heap property by recursively exchanging the key with one of a child. The number of steps performed by enqueue and dequeue operations are at most equal to the height of the tree, that is $O(\log n)$ where n is the number of nodes in the tree.

A binary search tree instead cannot guarantee an $O(\log n)$ upper bound. In fact,

inserting a sorted sequence of keys makes nodes arranged as in a linked list, since all nodes except the root are either left child or right child. In order to avoid this behavior, a set of re-balancing operations should be performed to maintain the height of the tree bounded to $\log n$. The splay-tree [28] is a binary search tree in which it is defined a *splay* operation that re-balances the tree. A dequeue consists in searching the minimum, that is always found by traversing left nodes. When a node without left child is found, it is the minimum, thus it is removed by connecting the right child to the parent and then a splay operation is invoked on the parent. Enqueue operation traverses the tree until the appropriate position is found in a leaf or in a node without the right child, then it connects the key according to the definition of binary search tree and finally applies a splay operation. The splay operation is a set of constant-time rotations involving at most three nodes with the objective to move accessed nodes near to the root. This guarantees that dequeue and enqueue operations are performed in $O(\log n)$ steps on average.

An alternative to balanced search trees is the skip-list [29]. A skip-list is a linked-list where nodes may maintain additional pointers to some successors that allow to skip intermediate nodes during a traversal. The number of additional pointers defines the *level* of the node. A node at level k maintains k pointers, where the i-th one points to the following node of level $j \geq i$. The level k of a node is chosen randomly during its insertion with probability p^{k-1}. This guarantees that 100% of nodes are at least at level 1, the $p \cdot 100\%$ at least at level 2, the $p^2 \cdot 100\%$ at least at level 3 and $p^{1-k} \cdot 100\%$ at least of level k. Therefore, reaching a given position depends mainly on the number of levels climbed during the traverse of the skip-list. Since the expected maximum number of levels is given by $\log_p(n)$, the same upper bound is given for enqueue operations. On the contrary, the minimum is always the first element of the list ensuring $O(1)$ for its retrieval.

2.3 Multi-list Based Pending Event Set

Multi-list based priority queues are generally implemented as a two-level data structure (Figure 2.3). The first level is an array accessed by using an index computed directly

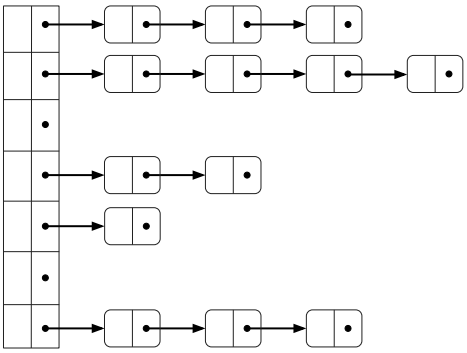


Figure 2.3: Visual representation of a multi-list object.

from the event timestamp and, due to the fact that more events can collide on the same index, a second-level data structure is used to resolve this conflict.

The calendar queue [30] is an array of ordered linked-lists, also called buckets. The idea is to partition the time axis in slots and then to assign them to each bucket in a circular fashion. In order to efficiently compute the minimum, it is paired with an index C, that identifies the time-slot containing the minimum, and with a counter, which maintains the number of events in the queue. Let B be the number of buckets and BW be the slot size, enqueuing an event e with timestamp T_e consists in inserting it into the i-th list of the array, where $i = \left\lfloor \lfloor \frac{T_e}{BW} \rfloor \mod B \right\rfloor$ and then increasing the counter of events. Finally, if $\lfloor \frac{T_e}{BW} \rfloor < C$, C is updated. The dequeue operation consists in returning `null` if the counter is equal to zero, otherwise it checks the c-th bucket, where $c = \left\lfloor C \mod B \right\rfloor$. If the bucket is non-empty and the first event e has timestamp $T_e \in \left\lfloor C \cdot BW, (C+1) \cdot BW \right\rfloor$, it removes e, decreases the counter and returns e, otherwise it increases C and repeats the check until an event is returned. The number of steps performed by a dequeue is constant if the current bucket is non-empty, otherwise a scan of the buckets is required. Since time slots are assigned in a circular policy a single scan of the whole array is enough to find the bucket containing the minimum, thus the search costs $O(B)$. On the other hand, the number of items per bucket can be $\frac{n}{B}$, but if we assume that the farther event from the current minimum is $75\%B$ time-slots forward, most of the enqueues are served in a quite empty bucket, leading to an $O(1)$ enqueue operation. To maintain the cost of a dequeue independent from the number of buckets B, the size of the array has to be changed according to the number of items in the queue. In particular when the number of

events is half the number of buckets, a new array half the previous size is allocated and each event is moved from old buckets to new ones. On the contrary, when the number of events is twice the number of buckets, the array is doubled. Anyhow, in order to guarantee that events are spread uniformly on buckets, the length of the time-slot is recomputed during a resize operation by observing the time separation among consecutive events with smallest priority. Since resize operations require a whole copy of all events, they perform $O(n)$ steps, but, since each item has been added paying a constant, the amortized cost is $O(1)$ for all operations.

The lazy queue, presented in [31], also adopts a multi-list based approach. The lazy queue consists in three data-structures: one sorted list (NF) for near-future events, one dynamic array of unsorted list (FF) for far-future events, one sorted list (VF) for very-far-future events. The idea is to split the time axis in three sections, each one assigned to a sub-structure of the queue. The bind is made by specifying an interval of time $[T_{NF}, T_{VF}]$ such that NF covers the time interval $[0, T_{NF})$, FF covers $[T_{NF}, T_{VF})$ and VF covers $[T_{VF}, +\infty)$. Assuming that D is the size of the array, the time interval covered by FF is split in D sub-intervals each one associated to a list. Given an event e with timestamp T_e, the enqueue operation:

- inserts into VF if $T_e > T_{VF}$;
- inserts into the i-th list of FF if $T_e \in [T_{NF}, T_{VF}]$, where $i = \lfloor \frac{T_e - T_{NF}}{D} \rfloor$;
- inserts into NF if $T_e < T_{NF}$.

If the NF structure contains N_{NF} events and VF contains N_{VF}, insertion costs $O(N_{NF})$ and $O(N_{VF})$ respectively, since they are sorted lists. On the contrary inserting in FF costs $O(1)$, because we add the new event at the head of an unsorted list. Assuming $N_{NF} << N$ and $N_{VF} << N$, where N is the number of events into the queue, the cost for the enqueue operation is $O(1)$. The dequeue operation is quite more involved, in fact, if the NF list is non-empty, its first element is removed and returned, otherwise the first non-empty list of FF array is sorted and moved into the NF and T_{NF} and T_{VF} are incremented by a factor equal to $\frac{T_{VF} - T_{NF}}{D}$. This leads to a $O(N_{FF} \log N_{FF})$ bound for dequeues, where N_{FF} is the average number of events per list of the FF structure. Thus under the assumption that $N_{FF} << N$, dequeues are performed in $O(1)$ steps. In

order to maintain the $O(1)$ bound for dequeue and enqueue operations, a set of resize operations of FF is defined:

- halving or doubling the number of lists;
- halving or doubling the distance between T_{NF} and T_{VF}

Changing the number of lists requires that each item in the FF is moved into a new FF leading to a $O(1)$ amortized cost. On the contrary, updating the size of the interval $[T_{NF}, T_{VF}]$ means inserting items in NF and VF. In order to do it efficiently the events are sorted and connected to NF and VF. The sorting has a $O(m \log m)$ cost, where m is the number of events moved from FF to NF or to VF, giving an average access time of $O(\log m)$. Assuming $m << N$ we can safely claim that every resize operation requires $O(1)$ steps in average.

The ladder queue [32] is an evolution of the lazy queue. It maintains the division of the time in three sections, individuated by two thresholds T_{NF} and T_{VF}, and maps them on three data-structures that the authors call Top, Ladder and Bottom. The Top structure is an unsorted linked list, the Bottom is a sorted list and the Ladder is a set of arrays of lists. The principle of operation is very similar to the lazy queue: the Bottom list contains events that are going to be dequeued, events in the Ladder are partially sorted, while the Top list contains unsorted very-far-future events. Differently from the lazy queue, the advancing of time proceeds in epoch. In fact, when the Bottom list is empty, the threshold T_{VF} is untouched and only T_{NF} is updated. T_{VF} is increased only when all events in $[0, T_{VF}]$ are dequeued. Assuming that the queue is empty an enqueue consists in connecting the new event in the Top list. The first dequeue moves events from the Top to the Ladder and individuates the thresholds T_{VF} and T_{NF} in the maximum and minimum timestamps of moved events. From this moment every event e with timestamp $T_e > T_{VF}$ will be connected to the Top list, otherwise either into Ladder or into Bottom. The Ladder is a set of arrays, called Rungs, and each list in a Rung is called a bucket. The idea of the Ladder is to focus resources on a predetermined interval of time. The interval of time covered by Ladder is subdivided in sub-intervals, that are processed one after the other. The first sub-interval is recursively split in smaller intervals until the first one contains few events. When a set E of events

is moved from Top to Ladder, the time between the minimum and maximum timestamp in E is computed and subdivided in $N = |E|$ intervals, each one associated to a distinct bucket of the first Rung. Thus an event $e \in E$ with timestamp T_e is inserted into the i-th bucket of the first Rung, where $i = \lfloor \frac{T_e \cdot N}{T_{VF} - T_{NF}} \rfloor$. Then the first non-empty bucket is searched and if it contains a number of events lower than a given threshold Max_R, its events are sorted and inserted into Bottom. If more than Max_R events are contained in the first non-empty bucket, a Rung with Max_R buckets is allocated and the time interval covered by the bucket of the first Rung is divided in Max_R sub-intervals, each one associated to a new bucket. The procedure is repeated until the first non-empty bucket of the last allocated Rung contains fewer events than Max_R. At this point they are moved into Bottom and a dequeue can remove and return the first event of the Bottom list. A successive enqueue observes the intervals covered by the Top and Bottom and decides where to insert the new event. Now inserting into Ladder requires a recursive analysis of the Rungs in order to find a bucket in the appropriate Rung. When the Bottom list is empty a new bucket is searched in the last allocated Rung. If they are all empty, the bucket is searched into the previous Rung and eventually it can be split with a new Rung. When Ladder and Bottom are empty an epoch is ended and events in Top are moved into the Ladder restarting the whole process. The authors explain that the number of Rungs is limited and dependent only on the priority increment distribution. This guarantees that every operation requires $O(1)$ steps in order to complete. In fact, an enqueue, that inserts into Top, add the new event at the head of the list. Insertion into Bottom requires a scan, but we can assume that the items into Bottom are few. Finally inserting into the Ladder requires a number of steps that is limited by the number of Rungs and inserting into a bucket cost $O(1)$, because it is an unsorted list. Finally a dequeue requires one step when the Bottom is non-empty, otherwise events have to be moved among different Rungs, but, since the number of Rungs is bounded and independent from the number of events, the operation is performed in $O(1)$ amortized constant time.

2.4 Non-blocking Pending Event Set

In a parallel event-driven program the pending event set is shared among multiple threads, thus its performance is more critical because it is a bottleneck for scalability. In fact each thread that wants to update the PES has to take the ownership of the whole data structure or of a part of it, blocking other threads. It means that also an efficient implementation of a pending event set can perform poorly under high contention. A strategy to resolve issues due to high contention is adopting a non-blocking implementation. Harris' Sorted Linked List presented in Section 1.5.2 can be used as a linear priority queue, but it guarantees an $O(n)$ bound for enqueues, thus, even if it is non-blocking and allows high concurrency, we expect that is asymptotically inefficient for a significant amount of events. In [33], the authors propose a linearizable lock-free implementation of skip-list structure, that is a valid solution offering $O(\log n)$ bound for enqueue and $O(1)$ for dequeue. However, it can handle events with identical priority by using few bits of the key for achieving uniqueness. On the contrary, the landscape of lock-free pending event sets, that guarantee constant access for both enqueue and dequeue, seems to be desolate. To our knowledge the unique result that tries to reach the goal is the work in [34], where a lock-free implementation of a ladder queue is presented. The authors simply exchange every list in the ladder queue with respective non-blocking counterpart. Anyhow we believe that is a partial result, since some relevant details, such as stability and ensured level of correctness, are omitted. For example, it is not explained if events are moved among buckets preserving FIFO order or if it is possible that a slow thread can insert an event in a bucket, whose events are just moved in a new Rung. In other words it is not explained if operations that update the global status of the ladder queue are lock-free, but, since the authors claim:

> "We propose the replacement of unsorted lists in Top, Rungs and Bottom with unsorted lock-free queues"

we assume that those operation are blocking. Moreover it is designed for systems that can handle events returned out-of-order, limiting the possible usages.

In Figure 2.4 is proposed a resuming of discussed priority queues for the pending

Implementation	Enqueue	Dequeue	Stable	Lock-free	Linearizable
Sorted Linked List	$O(n)$	$O(1)$	✓	✓	✓
Unsorted Linked List	$O(1)$	$O(n)$	✓	✓	✓
Binary Heap	$O(\log n)$	$O(\log n)$	✓	✗	-
Splay Tree	$O(\log n)$	$O(\log n)$	✓	✗	-
Skip-List	$O(\log n)$	$O(1)$	✓[1]	✓	✓
Calendar Queue	$O(1)$	$O(1)$	✓	✗	-
Lazy Queue	$O(1)$	$O(1)$	✓[2]	✗	-
Ladder Queue	$O(1)$	$O(1)$	✓	✗[3]	-

[1]Require using bits of the key for *tie-braking*.
[2]Require a stable sorting.
[3]We are assuming that operations on the global status in [33] are blocking.

Figure 2.4: Comparative table of discussed pending event set implementations.

event set problem.

Chapter 3

Non-Blocking Priority Queue For Pending Event Set

In this chapter we introduce a lock-free data-structure for Pending Event Set management. Since we want to reach an amortized constant cost for each operation, we have chosen a multi-list based approach (see Section 2.3) to arrange events. Moreover this strategy offers a natural way to limit concurrency only on a subset of items stored in the data structure. The time axis is subdivided in equal slots and each list, that we call bucket, is associated with a particular time slot $[a, b)$ and it is able to contain only events with timestamp $T \in [a, b)$. The length of time-slots is called bucket width and generally should change over time in order to control the average number of items per bucket. It means that also the number of buckets should be variable in order to cover a fixed interval of time. In our solution we focus only on the second point, because the optimal bucket width is related to a several number of factors, including the number of buckets and the policy used to insert items into them. Moreover, since we do not know a sequential implementation that can be easily transformed in a non-blocking solution with constant time access, we cannot immediately take advance from the extensive research on multi-list based priority queues. In fact changing the bucket width means change the strategy used to refer buckets. In a non-blocking solution this is hard, because, since we cannot block threads and change the bucket width once for all, we have to support two or more bucket addressing at the time. Therefore it is not necessary that optimal choices in a sequential execution

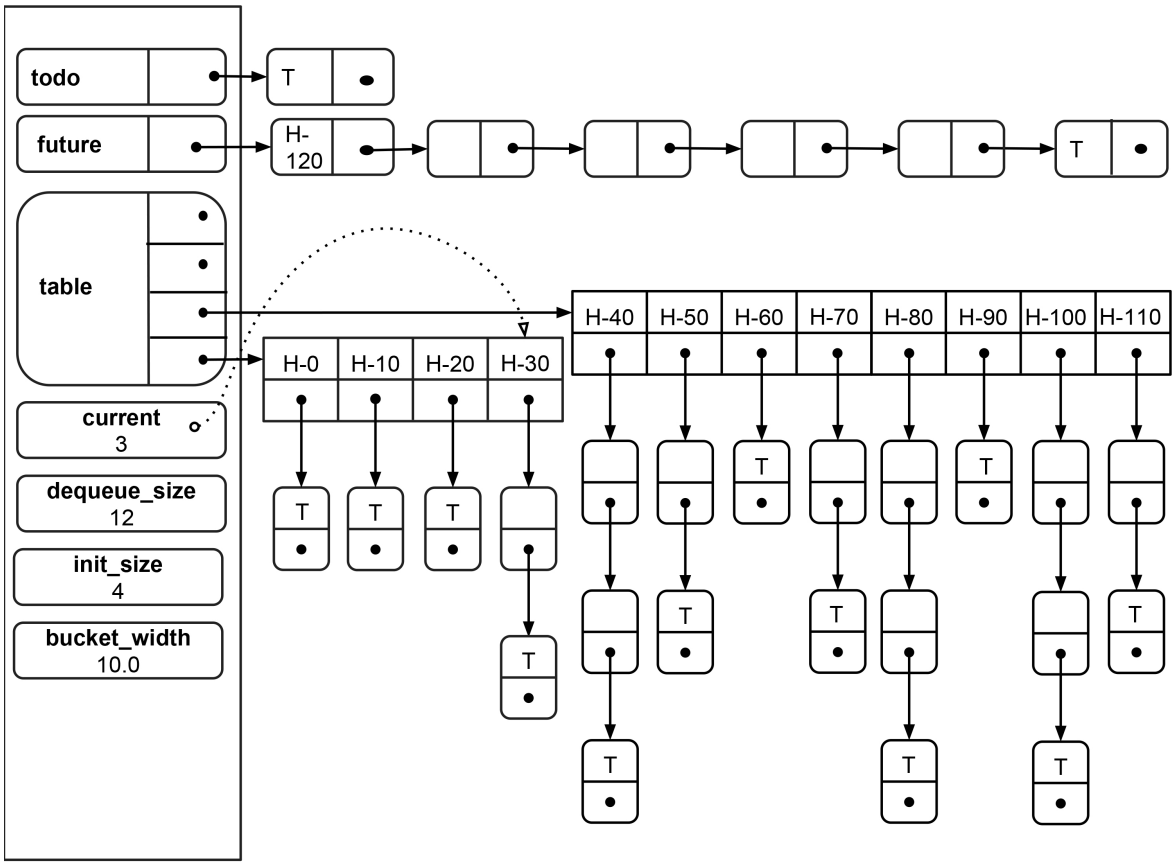


Figure 3.1: Visual representation of our non-blocking priority queue.

are feasible in a non-blocking solution.

A visual representation of the whole data-structure is proposed in Figure 3.1. The buckets are arranged in a dynamic array `table` in order to allow accesses based on indeces. The queue uses some variables to keep in memory useful information:

- `current` stores the index of the bucket which contains the minimum timestamp;
- `init_size` is the initial number of buckets in the queue;
- `bucket_width` is the width of a bucket, i.e., the length of the time interval covered by each bucket.

Since each bucket covers a fixed interval of time, at start-up the whole set of lists covers a time interval equal to $[0, \texttt{bucket\_width} \cdot \texttt{init\_size})$. In order to cover events in $[\texttt{bucket\_width} \cdot \texttt{init\_size}, +\infty)$, it is paired with an overflow data-structure pointed by `future` and eventually by `todo`. For a more readable explanation, we call `future` and `todo` the stack pointed by `future` or `todo` respectively. The overflow structure

```
NBPQueue{
  pointer<Stack> future;
  pointer<Stack> todo;
  pointer<HarrisSet> table[32];
  integer current;
  integer dequeue_size;
  integer init_size;
  double bucket_width;

  NBPQueue NBPQueue(integer size, double width){
   1: new ←NBPQueue()
   2: new.todo ←Stack(0)
   3: new.future ←Stack(size)
   4: new.init_size ← size
   5: new.dequeue_size ← size
   6: new.bucket_width ← width  return new
}
```

```
Stack{
  pointer<Node> top;
  integer enqueue_size;

  Stack Stack (integer index)
   1: new ←new Stack()
   2: new.top ← null
   3: new.enqueue_size ← index return new
}
```

Figure 3.2: Non-blocking priority queue object.

is a stack associated with a variable `enqueue_size` representing the left limit of the covered timestamps in $[\texttt{bucket\_width} \cdot \texttt{init\_size}, +\infty)$. The array maintains, in a variable `dequeue_size`, the right limit of the time interval covered by all buckets. `todo` is initially set to `null`. At start-up, we have `init_size` = `dequeue_size` = `enqueue_size`. The structure of our Pending Event Set is shown in Figure 3.2.

During the life of the queue, the following operating rules are maintained:

1. dequeues return an event with timestamp in $[0, \texttt{dequeue\_size})$;
2. the stack pointed by `future` contains events in $[\texttt{enqueue\_size}, +\infty)$;
3. enqueues can only decrease `current`;
4. dequeues can only increase `current`.

When an event e with timestamp T_e is enqueued, its index is computed as $i_e = \lfloor \frac{T_e}{\texttt{bucket\_width}} \rfloor$ and if $i_e < \texttt{enqueue\_size}$ the event is inserted into the corresponding i_e-th bucket, otherwise it is pushed in `future`. Finally if $i_e \leq \texttt{current}$, the index of the current minimum is updated.

On the other hand, the DEQUEUE procedure checks that the bucket at index `current` is not empty. In this case the first node of the list is removed and then

returned, otherwise `current` is incremented by one and the check is repeated until a non-empty bucket is found or the value of `current` is equal to `dequeue_size`. If the last condition is true and `future` is empty, we can return `null`, because the queue appears to be without events. Conversely, if the overflow structure has some items, it contains the new minimum. Thus we have to move items from `future` into the array. The first step consists in doubling the number of buckets in the queue. At this point a new enqueue of an event e', such that $i_{e'} \leq 2 \cdot$ `enqueue_size`, could be served by a new allocated bucket. To allow this without blocking, we store the current `future` in a temporary storage `todo` and we exchange the overflow structure with a new one, that covers timestamps in $[2 \cdot$ `bucket_width` $\cdot$ `enqueue_size`$, +\infty)$. Clearly the new `future` has an `enqueue_size` that is the double of the previous one. Now new enqueues can be served coherently, while dequeues have to move every event from `todo` into `table`. When `todo` is empty, `dequeue_size` is updated to its double and dequeues can restart from `current` = `dequeue_size`. At this point we have restored the initial condition `dequeue_size` = `enqueue_size`.

3.1 Algorithm

The proposed algorithm is based on a set of linearizable and lock-free data-structures. The building blocks are the Harris' Sorted Linked List and Treiber's Stack, to which we have added some additional feature useful to our purpose.

Harris' Sorted Linked List In order to support events with equal timestamps, we have augmented the SEARCH routine of the linked list in order to retrieve a right node such that has a key greater than or equal to the search key. This is obtained by changing the condition on the right node of the search routine (line S13). Anyhow we have left the capability to invoke the original search by adding a parameter that allows us to switch between the two versions. In particular, given a search key k, the original version can be used by invoking SEARCH(k, $<$) and it return a right node with key greater than or equal to k. On the contrary, a right node with timestamp strictly greater than k is obtained by using SEARCH(k, $\leq$).

```
boolean Stack.tryPush(Node n){
 1: tmp ← top
 2: n.next← tmp return
 3:        ¬ISMARKED(tmp) ∧ CAS(&top, tmp, n)

boolean Stack.denyPush(){
 1: repeat
 2:     tmp ← top
 3: until CAS(&top, UNMARK(tmp), MARK(tmp))
```

```
Value Stack.pop(){
 1: repeat
 2:     tmp ← top
 3:     tmpNoMark ← UNMARK(tmp)
 4:     new ← tmpNoMark.next
 5: until CAS(&top, tmp, MARK(new)) return
    tmp.key
}
```

Figure 3.3: Augmented Treiber's Stack algorithm.

Treiber's Stack We have added a new procedure DENYPUSH, that denies any enqueue to succeed. This is achieved by marking the `top` field as it is done the for logical deletion in the Harris's list. After that DENYPUSH completes, any push attempt fails to add an event and only POPs are allowed. Consequently we have defined a TRYPUSH routine that tries to insert a node into the stack with a `CAS` only if the `top` field is not marked and it returns true if and only if the swap succeeds. This allows us to ensure that POP and TRYPUSH cannot alternate each other after a completed DENYPUSH. Finally, our stack maintains the left limit of the time interval that it covers in a variable `enqueue_size`. A complete view of the new implemented stack is shown in Figure 3.3.

As explained in the previous section, the number of buckets in the queue increases over time. In order to maintain an indeces-based access, they are stored in a dynamic array (`table`) that never moves already allocated blocks as in [19] and discussed in section 1.5.3. Thus `table` is a two level array. The first array has a fixed size and contains pointers to contiguous memory blocks, that contain the `head` node of buckets.

ENQUEUE (Figure 3.4) works in two phases. The first phase consists in connecting the node to the structure. The INSERT routine retrieves first the `future` stack and checks the `enqueue_size` in order to discover where the node has to be connected. If it is greater than or equal to the linear index i of the event to be inserted, computed as $i = \lfloor \frac{timestamp}{\texttt{bucket\_width}} \rfloor$, then it attempts to connect the new item to the stack with a TRYPUSH and returns if it succeeds, otherwise it restarts from the beginning. The try loop ends if either the TRYPUSH succeeds or the condition $i <$ `enqueue_size` is

```
NBPQueue.enqueue(event e)
1: index ← ⌊ newNode.t / bucket_width ⌋
2: newNode ← new node(e)
3: INSERT(newNode, index)
4: FLUSHCURRENT(index)
```

```
NBPQueue.insert(node newNode, integer index)
1: repeat
2:     tmp ← future
3:     tmpSize ← tmp.enqueue_size
4:     if tmp.TRYPUSH(newNode) then
5:         return
6:     endif
7: until index < tmpSize
8: bucket ← table[h1(index)][h2(index)]
9: repeat
10:    ⟨leftNode, rightNode⟩ ← bucket.SEARCH(newNode.t, ≤)
11:    newNode.next ← rightNode
12: until CAS(&leftNode.next, rightNode, newNode)
```

```
integer NBPQueue.h1(integer index)
1: return (ibsr(index) − ibsr(init_size) + 1) & (-(index ≥ init_size))
```

```
integer NBPQueue.h2(integer index)
1: returnindex & (∼ ( (index ≥ init_size) << ibsr(index)))
```

```
NBPQueue.flushCurrent(integer n)
1: repeat
2:     old ← current
3:     ind ← old >> 32
4:     if n > ind then return
5:     end if
6: until CAS(&current, old, (n << 32) | ABAMARK())
```

Figure 3.4: Non-blocking ENQUEUE for pending event set.

true. In this case we compute the location of the i-th bucket and we insert the node using the augmented search of the Harris' Sorted Linked List. This guarantees that events with same timestamps are inserted in a FIFO order into the list. At this point the phase of insertion is completed and the event is connected either to `future` or to `table`. The second phase guarantees that `current` points to the bucket containing the minimum. This is done by FLUSHCURRENT procedure that tries to exchange the `current` variable with a `CAS` until either it succeeds or i is strictly greater than the current value of `current`.

```
event NBPQueue.dequeue()
 1: oldCur ← current
 2: index ← oldCur >> 32
 3: min ← table[h1(index)][h2(index)]
 4: minNext ← min.next
 5: ⟨_, right⟩ ← SEARCH(min.t, min, <)
 6: candidate ← right
 7: rightNext ← right.next
 8: index2 ← current >> 32
 9: if candidate ≠ tail then
10:     if ¬ISMARKED(rightNext) ∧ index2 ≥ index ∧
11:                                   CAS(&candidate.next, rightNext, MARK(rightNext)) then
12:         return candidate.event
13:     else
14:         continue
15:     end if
16: end if
17: index ← index + 1
18: tmpSize ← dequeue_size
19: if index = tmpSize then
20:     tmpFut ← future
21:     eSize ← tmpFut.enqueue_size
22:     if tmpSize = eSize ∧ index2 ≥ index − 1 ∧ tmpFut.next= null then
23:         return null
24:     endif
25:     if ¬EXPANDARRAY(tmpSize) then
26:         continue
27:     endif
28: endif
29: CAS(&current, oldCur, (index << 32) | ABAMark() )
30: goto 1
```

Figure 3.5: Non-blocking DEQUEUE for pending event set.

DEQUEUE (Figure 3.5) starts to search the minimum in the bucket corresponding to `current`. The standard search offered by the Harris' List is used by invoking SEARCH(`bucket_width` · `current`, <). This ensures that the left node is always the head of the list, while the right node has the minimum key in the bucket. Once the minimum is identified. DEQUEUE restarts if the `current` is decreased by another thread, otherwise it tries to logically remove the right node by marking it with a `CAS`. If it succeeds, DEQUEUE completes and returns after searching again the minimum in the bucket in order to remove marked nodes. However, it possible that the bucket is empty and then the right node is `tail`. In this case we have to update `current` with a `CAS`, making it point to the next bucket. If the `CAS` fails, it means that some thread has inserted a node in a bucket that precedes the next one, thus we start from beginning.

Now four cases are possible:

a) `current` $<$ `dequeue_size` $- 1$;

b) `current` $=$ `dequeue_size` $- 1$, while `future` is empty and its `enqueue_size` $=$ `dequeue_size`;

c) `current` $=$ `dequeue_size` $- 1$, while `future` is non-empty and its `enqueue_size` $=$ `dequeue_size`;

d) `current` $=$ `dequeue_size` $- 1$, `future` has `enqueue_size` $\neq$ `dequeue_size`.

Case a) implies that the next time interval is covered by `table`, thus the next bucket is candidate to be checked for searching the minimum. This is achieved by incrementing `current` with a `CAS` and restarting the operation.

Case b) means that we have passed all buckets in `table` that covers the interval $[0, \texttt{dequeue\_size} \cdot \texttt{bucket\_width})$ and no items are stored in the overflow data structure associated with time interval $[\texttt{dequeue\_size} \cdot \texttt{bucket\_width}, +\infty)$, thus we can safely return `null` to indicate that the queue is empty.

Case c) occurs when `current` points to the last bucket in `table` and, at the same time, the `future` is non-empty. It means that `table` can be expanded and every item in `future` can be inserted into appropriate buckets. Before moving nodes, we verify that the `enqueue_size` associated with `future` is equal to the `dequeue_size` read when the operation is started. This avoids that two expansion can occur simultaneously. After this check, we invoke the routine expandArray, that returns true if `dequeue_size` is doubled at the ending of its invocation. If it returns false then we restart from the beginning, otherwise we are sure that the `table` covers the next time interval and we can increment `current` with a `CAS`. Independently from the result of this `CAS`, we restart from the beginning.

Case d) allows to detect whether an expansion is already occurring, thus expandArray is executed in order to help any thread that is execution an expansion.

Finally dequeue completes only if it succeeds to mark a node or it meets the empty condition (case b)).

expandArray (Figure 3.6) has two roles: extending `table` and moving the elements from the overflow structure to `table`. The first phase occurs only if

```
boolean NBPQueue.expandArray(oldSize)
 1: tmpFut ← future
 2: if tmpFut.enqueue_size = oldSize then
 3:     newBlock ← newarray[oldSize]
 4:     if ¬ CAS(&table[h1(oldSize)], null, newBlock)
 5:         delete newBlock
 6:     endif
 7:     tmpTodo ← todo
 8:     if tmpTodo.enqueue_size < oldSize then
 9:         CAS(todo, tmpTodo, tmpFut)
10:     end if
11:     tmpTodo.denyPush()
12:     CAS(future, tmpFut, new future(2 · oldSize))
13: end if
14: while UNMARK(todo.next) ≠ null do
15:     node ← todo.pop()
16:     insert(node)
17: end while
18: CAS(&dequeue_size, oldSize, 2 · oldSize)
19: return oldSize < dequeue_size
```

Figure 3.6: expandArray routine algorithm.

`enqueue_size` associated with the current `future` is equal to `dequeue_size` of the invoking queue. The expansion of the array is performed by allocating the new required block and connecting it to the first level array with a single-shot `CAS`, because the old value is `null`. Before the stack could be emptied, we have to communicate to all threads that an expanding phase is occurring. This is obtained by making the reference `todo` pointing to the current future with a single-shot `CAS`. The exchange is performed only if `enqueue_size`, stored in the stack currently pointed by `todo`, is strictly lower than `dequeue_size` seen when dequeue is invoked. This avoids that `todo` is reset to its initial condition, while other threads are moving events. Now `table` is capable to store items in an increased interval of time, but this information is not published yet. Thus we exchange `future` with a new stack such that its `enqueue_size` is equal to $2 \cdot$ `dequeue_size`. Since the size of the array is always increasing, it is ensured that exactly one `CAS` can successfully update `future`, because new expansions cannot occur until `dequeue_size` is updated. When the new stack is installed, enqueue sees a wider `table` than dequeue. After that this condition is met, the stack referred by `todo` can be emptied with a sequence of consecutive pops, but we have to invoke denyPush first. Preventing that tryPush can succeed allows to reach a stable condition in which

the stack is empty. In fact, we have ensured that a very slow thread cannot succeed to push an event after the stack is detected as empty. Finally, when this condition is verified, the current `dequeue_size` is doubled with a single-shot `CAS`, restoring the condition `enqueue_size` = `dequeue_size`.

3.2 Guaranteed Properties

In this section we will show the guarantees that are provided by our data structure. We start showing an event e cannot be lost. With the term lost we mean that an enqueued event e either it is never dequeued or it requires an insertion of an event e' such that $T'_e \leq T_e$, because the advancing of `current` has left behind e that will be dequeued if and only if a new enqueue moves back `current`. Let D be the value of `current`. If the timestamp T_e associated to e is such that $T_e \leq D \cdot$ `bucket_width`, ENQUEUE inserts e in `table` and cannot terminate until `current` is updated, ensuring that the node will be dequeued. If $T_e > D$, then it can be connected either to `table` or to `future`. In the first case the node will be dequeued, because each bucket cannot be skipped, since `current` is increased by one at a time. If an insertion in the current bucket occurs after it is detected as empty, ENQUEUE tries to update `current` even if it is pointing to the same bucket. This ensures that either an increment of `current` fails or the ENQUEUE retries updating `current`. On the other hand, when a node is connected to `future`, it can be lost if and only if `todo` is lost. In fact the augmented stack allows a node to be enqueued until the first DENYPUSH succeeds, thus a thread that fails to connect for this reason, will retry to insert either in a new stack or into `table`. After that it succeeds to insert the node into the overflow structure, it will be moved into `table` when `future` will become a `todo` stack and it will be emptied. A stack pointed by `todo` cannot be lost because it is exchanged once during an expansion phase. This is guaranteed by:

- swapping the old `todo` if its `enqueue_size` is strictly lower than the new one;
- the fact that only threads in DEQUEUE procedure can participate to the expansion;
- those threads cannot exit from EXPANDARRAY until the current `todo` is empty.

3.2.1 Linearizability

In order to show the assumptions that make our algorithm linearizable, we first give a formal definition of the abstract PES and of our data structure (NBPQ).

Definition 3.1. *A Pending Event Set is a set of events on which are defined two procedures: Enqueue(e) (denoted as E) and Dequeue() (D). The state S(t) at time t of a PES is the set of events in the structure at that time. Let O_t be a procedure in $\{E, D\}$ executed at time t; $A : O_t, B$ be the semantics of an procedure execution where A is the conditional state before O_t and B the resulting state. We have:*

$$e \notin S(t_-) : E_t(e), S(t_+) = S(t_-) \cup \{e\} \tag{3.1}$$

$$e = \min\{S(t_-)\} : D_t() = e, S(t_+) = S(t_-) \setminus \{e\} \tag{3.2}$$

$$S(t_-) = \emptyset : D_t() = \perp, S(t_+) = S(t_-) \tag{3.3}$$

Definition 3.2. *An event is said to be connected to NBPQ if and only if it is encapsulated in an unmarked node that is stored either in one bucket or in the overflow stack pointed by* `future` *or* `todo` *.*

Definition 3.3. *A state R of NBPQ is a pair $\langle C, L\rangle$ where C is the left limit of the time interval of the bucket pointed by* `current` *and L is the set of connected events.*

Lemma 1. *Any state transition of NBPQ is atomic.*

Proof. The operations that can change a given status $R = \langle C, L\rangle$ of a NBPQ are those that modify either the overflow stack or a bucket or the `current` field. In particular any change on `current` is executed by a successful `CAS` in a FLUSHCURRENT or in a DEQUEUEinvocation. This updating atomically changes the status R to $R' = \langle C', L\rangle$, where C' is the new value of `current`. The Harris' Sorted Linked List is a linearizable object, thus its INSERT and DELETE appears to be executed atomically in a linearization point, that respectively adds an event e and or marks it. The new status after their ending will be $R' = \langle C, L \cup \{e\}\rangle$ or $R' = \langle C, L \setminus \{e\}\rangle$. A TRYPUSH on the stack adds a new event e with an atomic successful `CAS`. The new event cannot be lost, thus a new NBPQ state R' is created such that $R' = \langle C, L \cup \{e\}\rangle$. Since every new possible state is created at an indivisible point in time, every state transition of NBPQ is atomic. □

Lemma 2. *The* DEQUEUE *of a node n, which has never been in an overflow stack, appears to be executed atomically at the atomic read that individuates n.*

Proof. A node n is dequeued by a DEQUEUE d after it is marked with a `CAS`. Before this operation, an atomic read r on the `next` field of the head node p is performed or, equivalently, a `CAS` that make the head node and n adjacent succeeds. A concurrent DEQUEUE d' can mark either a previous node of n and its read happens before the read of d or a successor of n and its read happens after the read of d, since we know that d has marked n. A concurrent ENQUEUE i on the same bucket adds a node n' with a `CAS`. If this update happens in time before r, it follows that $n = n'$ since in r they was adjacent, otherwise n' is a successor of n. A concurrent ENQUEUE in a bucket after the current one is irrelevant since it is inserting an event with timestamp strictly greater than the minimum. A concurrent ENQUEUE j which inserts a node in a bucket k before the current one completes only if `current` assumes a value lower than or equal to k. Since d, before trying to mark n, checks that `current` is unchanged, j is still pending when r occurs. Thus the DEQUEUE d appears to be executed atomically at the read r. □

Lemma 3. *The* DEQUEUE *operation that returns* `null` *appears to be executed atomically at the atomic read that has detected the last bucket as empty.*

Proof. A bucket is empty when head and tail nodes are adjacent. The last bucket is seen as empty with an atomic read from Lemma 2. After this read, a reference to `future` is taken with a second atomic read. If `future` allows push operations, but, it results empty and its `enqueue_size` is equal to the `dequeue_size`, the stack was empty also at the first read, because events cannot be lost and after a DENYPUSH, no TRYPUSH can succeeds and POP is invoked only after a successful DENYPUSH. □

It follows that the evolution of a NBPQ can be represented as a series of points on the real-time axes.

Definition 3.4. *Let $\mathbb{P}$ be the domain of NBPQ states, $\mathbb{S}$ be the domain of PES states, $R = \langle C, L \rangle$. $\pi : \mathbb{P} \mapsto \mathbb{S}$ is a mapping function such that:*

$$\pi(R) = \{e \in L \wedge T_e \geq C\} \tag{3.4}$$

Lemma 4. *The state transition of a PES represented with the mapping function π from states of a NBPQ are atomics.*

Proof. Let $R = \langle C, L \rangle$ be the current state of REP and let $S = \pi(R)$ be the state of PES. We consider first the ENQUEUE operation of an event e with timestamp T_e. We can distinguish two cases:

a) $T_e \geq C$: the node is connected with a `CAS` either to a bucket or to the top of overflow stack, giving $R \to R'$, where $R' = \langle C, L \cup \{e\} \rangle$. $S = \pi(R)$ and $S' = \pi(R')$, thus the successful `CAS` produces a state transition $S \to S'$ of PES, that is atomic.

b) $T_e < C$: the node is connected with a `CAS` giving $R \to R'$, where the new state $R' = \langle C, L \cup \{e\} \rangle$, but $\pi(R) = \pi(R')$. Then ENQUEUE tries to update `current` in order to point to the bucket B_e containing the event e. Let C_e be the left limit of the time interval associated with B_e. The update of `current` is performed with a `CAS` so atomically we have $R' \to R''$ where $R'' = \langle C_e, L \cup \{e\} \rangle$. If more than one ENQUEUE $E', E'', \ldots, E^{(n)}$ inserts respectively the events $e', e'', \ldots, e^{(n)}$ in the interval $[0, C)$, they first connect their events with a `CAS` and then they try to update the `current` with their $C_{e^{(i)}}$. Let $E^{(k)}$ be the first ENQUEUE that successfully updates `current` with $C_{e^{(k)}}$ by a `CAS` c and d be the linearization point of the first DEQUEUE after this update. For any $E^{(i)}$, that has $C_{e^{(i)}} \in [0, C_{e^{(k)}})$ nothing is changed, thus they are recursively in case b). Those of the remaining, that have connected their event after the point c, fall in case a). It means that the considered ENQUEUEs have connected their events before point c and have $C_{e^{(h)}} \in [C_{e^{(k)}}, C)$. When c is performed, all $E^{(h)}$ are pending and the represented PES skips atomically from the state $S = L$ to $S' = L \cup \{e', e'', \ldots, e^{(n)}\}$ that will be read by d. Since d occurs atomically from Lemma 2, it is always possible to find a point $p^{(h)}$ between c and d for each considered ENQUEUE $E^{(h)}$, such that it appears to be executed atomically at $p^{(h)}$. The ordering of those point is the same of the ordering in which the `CAS`es, that have connected events, are executed. This creates an order that is compatible with FIFO during DEQUEUE operations of events with identical timestamps.

Finally, DEQUEUE removes nodes only from the bucket pointed by C and, since every dequeue is atomic from Lemma 2, the state transition is atomic. □

Lemma 5. DEQUEUE *from a PES represented with the mapping function* π *on a NBPQ returns always the minimum, until no expansions occur.*

Proof. Let $R = \langle C, L \rangle$ be the state of NBPQ. Every dequeue from a NBPQ returns an event e with timestamp T_e that is the minimum in the bucket associated to C. From the definition of π, $T_e \leq T_{e_i}$, where T_{e_i} is the minimum timestamp of any $e_i \in \pi(R)$. □

Theorem 1. *Our representation is linearizable until no expansions occur.*

Proof. Any operation appears to take effect at one point. Thus the realtime order is preserved because: those points are between the invocation and the ending of any routine; they take place in an atomic instruction that defines a realtime order on concurrent procedures. Since every dequeue returns the minimum from Lemma 5, every history is valid. □

In order to be linearizable, we have to guarantee that any DEQUEUE, which requires an array expansion, returns events maintaining the realtime order. The proposed algorithm considers the expansion as terminated when the overflow stack pointed by `todo` is read as empty, i.e., the top variable is pointing to `null`. When this condition is met, the `dequeue_size` is updated and DEQUEUE can continue and analyze the next bucket. It means that, if a very slow thread A takes the last element from the stack, making it empty, a very fast thread B can enqueue and dequeue an event before A has inserted its event into `table`, violating real-time ordering. Moreover, if A is moving a node inserted previously by B, sequential consistency is violated too, because B can dequeue events with an order that is not compatible with its own history. The problem arises, because "the overflow stack is empty" and "all events are into `table`" are two separate conditions that happen at two distinct points in time, leading to a time window of vulnerability. If we can guarantee that no EXPANDARRAY can complete until every element in the stack is inserted in `table`, we obtain a linearizable algorithm. This is achievable by removing an event from the stack if and only if a copy is inserted in `table`. Since inserting events in `table` requires a scan of all events with same timestamp into the bucket, we can check if a copy is already inserted. The same test cannot be done when moving objects from `todo` to `future`, leading to a waste of memory when an event is copied multiple times on several `future`s. Moreover we can avoid this behavior by setting a flag to useless replicas already inserted in `future`. In fact, when a thread A successfully inserts a replica, then it executes a `CAS` on the top. If the exchange

fails, it means that some other thread B has performed a successful `CAS` with a POP (no PUSH can succeed after DENYPUSH) and thus it has successfully inserted a replica. It means that the replica created by A is useless and its flag can be set safely. This approach ensures that any DEQUEUE cannot scan the first bucket of the new block until all elements in `todo` are inserted into a new `future` or `table`.

3.2.2 Lock-Freedom

In order to show that our algorithm guarantees lock-freedom, we consider only retry-loop sections. An ENQUEUE consists in insert an event either in a bucket or in a stack and then trying to update `current`. Inserting in a bucket is lock-free because it is guaranteed by the Harris' Sorted Linked List. Inserting in the stack is a lock-free operation, because if it fails it means that someone have inserted an event and thus makes progress, or it has performed a DENYPUSH. If DENYPUSH fails, it means that some thread has inserted a node or has marked the top field. Finally, if it fails to insert in the overflow stack, it succeeds to insert in the `table` eventually, since it covers an ever increasing time.

The retry-loop in EXPANDARRAY invoked by a DEQUEUE, is lock-free because it retries until the stack is empty, but after the first successful DENYPUSH, no event can be inserted, thus the number of events in the stack is bounded. When a DEQUEUE fails to mark a node, it means that someone makes progress by inserting a new event or marking the node and thus completing a DEQUEUE. The update on `current` is performed by both ENQUEUE and DEQUEUE. If an update fails because of a successful ENQUEUE, this last one completes. Otherwise only DEQUEUEs make progress and thus they eventually will meet the empty queue condition.

3.2.3 Asymptotic Cost

The ENQUEUE operation consists of computing an index, inserting either in one bucket of the array or in the overflow structure and updating a value. The overflow structure is a stack with some augmented facility, but it guarantees $O(1)$ cost per operation. Inserting in a ordered list means that we have to scan the elements contained in it. Thus

the worst case is $O(n)$, but with a good choice of the bucket width, we aspect $O(k)$ on average where k is the average number of events in a bucket. The DEQUEUE operation costs $O(1)$ when the bucket containing the last minimum is non-empty. Otherwise a scan for the next non-empty bucket is performed, but the number of checks is unrelated to the number of events in the queue. Finally, moving elements from the overflow data structure to the array is performed in $O(c \cdot k)$ steps where c is the number of elements to be moved. Each of the c elements was added paying a constant cost (insert in a stack), that leads to an amortized constant cost for the whole operation.

3.2.4 Stability

Our implementation does not guarantee that events with the same timestamp are dequeued with the same order in which they were enqueued. When an event e is inserted in a bucket, the augmented SEARCH ensures that the right node is strictly greater than the timestamp of e, while the left node can be equal. In this case the events are inserted in same order in which the successful `CAS`es appear in time. Since DEQUEUE returns the first event in a bucket a FIFO order is ensured, until no events are connected to the overflow stack. In fact, when they are moved from the stack to a bucket, ordering information is lost. A simple strategy to avoid this could be inserting a counter into each node. When an event e is inserted in the overflow stack, the counter of e is updated to the counter of current `top` plus one. Since a node is connected only if the `CAS` is successful, it is ensured that counters increase monotonically over nodes. At this point insertion order can be conserved among events contained in the overflow stack. When events are moved from the stack to a bucket, those with identical timestamp are inserted according to their counter, where an uninitialized counter equals to infinity.

3.3 Dealing With ABA Problem

As shown in section 1.2.1, `CAS` is not able to distinguish if a memory location is updated with some previous value or it is unchanged and this can lead to an undefined behavior. The augmented stack (Figure 3.3) is ABA safe because it is not possible to reinsert a popped node, given that every TRYPUSH fails after the first completed DENYPUSH.

However nodes in `table` and in a stack must be protected with some form of garbage collection system in order to prevent a premature releasing. As shown in Section 1.6, a garbage collector can protect algorithms from ABA problem resulting from memory reusage. In Section 3.4 we present our memory reclamation system.

On the contrary, the `current` field must be protected with an appropriate mechanism, because it is an index that could assume old values at any time. For this reason, we rely on a 64 bit `CAS`, which is available on almost any machine, and we use 32 bits for storing the index and the others 32 bits as a signature of the update. The signature is generated using the bijective *Cantor pairing function*:

$$Cantor : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$Cantor(a, b) = \frac{(a + b)(a + b + 1)}{2} + b$$

that maps two integers on a unique natural number. The parameters used are the thread id *tid* and a private counter *ctr* that is incremented upon each successful update of `current`. Clearly the counter and the result can overflow leading to a possible collision, but their period should be long enough (about 2^{32} iterations for the result and 2^{64} for the counter) to guarantee that also a very slow process has a more recent version of the `current`.

3.4 Garbage Collection

As explained in Section 1.6, a non-blocking dynamic memory object requires a careful strategy to safely reclaim the memory. Moreover, adopting a garbage collection mechanism makes non-blocking objects ABA safe for free when reusing memory. Since a Pending Event Set is a data-structure that handles a particular kind of objects, we have implemented a simple garbage collector, which takes advantage from the association between events and logical time. In fact, assuming that events are generated by a stable system, values of timestamps should increase globally. This guarantees that after a certain amount of wall-clock time, events before a given time instant T are never enqueued. If it is true that no events with timestamp $T_e < T$ are inserted

anymore, we can assume that all buckets i such that $i \cdot \texttt{bucket\_width} < T$ are no longer accessed and thus no thread hold a reference to any event that was stored in those buckets. When nodes are disconnected from our queue, they are inserted in a thread private list, that is scanned by a procedure PRUNE that reclaims every event associated with a timestamp strictly lower than the threshold parameter T.

Chapter 4

Experimental Evaluation

The objective of following discussion is to show the results of our work and to point out its strengths and limitations in order to understand what should be the guidelines to implement a non-blocking pending event set with constant time access, that does not require an a priori knowledge of priority increment distributions.

Evaluation Program. The test program is a parallel one, in which each thread repeatedly enqueues or dequeues an event with a given probability P_E and P_D respectively, such that $P_E + P_D = 1$. Each thread maintains a local variable `local_time` representing the timestamp of the last dequeued event. A new event is associated with a timestamp obtained by adding a value `jump` to `local_time`. The `jump` represents the priority increment and it is obtained according to three different probability distributions with mean inter-arrival time $\mathbb{E}[T] \in \{1, 10, 50\}$. Increasing the mean value allows us to understand the behavior of our solution for several choices of bucket width.

The program ends when the total number of operations performed (which is independent of the actual number of parallel threads) reaches a given threshold $\#OPS = 1280000$. This guarantees that every thread works for the whole experiment with the highest concurrency. All tests are performed with a number of threads varying from 1 to 32.

The used distribution are uniform, triangular and exponential ones, computed as in Figure 4.1. The algorithm of our experiments is shown in Figure 4.2.

Probability Distribution	Formula
$Uniform$	$2\mathbb{E}[T]\cdot\texttt{rand}$
$Triangular$	$\frac{3\mathbb{E}[T]}{2}\cdot\sqrt{\texttt{rand}}$
$Exponential$	$-\mathbb{E}[T]\cdot\ln(\texttt{rand})$

Figure 4.1: Probability Distributions used in evaluation tests and their formula given the mean value $\mathbb{E}[T]$.

```
experiment(Value v){
 1: counters[i] ← 0
 2: for parallel i ← 0 to N do
 3:     repeat
 4:         if rand(Uniform) ≤ p_D then
 5:             local_time ← DEQUEUE()
 6:         else
 7:             ENQUEUE(local_time + rand(distribution))
 8:         end if
 9:         counters[i] + +
10:         sum ← 0
11:         for j ← 0 to N do
12:             sum ← sum + counters[j]
13:         end for
14:     until sum < #OPS
15: end for
}
```

Figure 4.2: Experiment algorithm.

The proposed non-blocking priority queue (NBPQueue) is initialized with a `bucket_width` equal to 1 and the `init_size` equals to 32768. The PRUNE routine is invoked periodically by a thread every 5000 operations completed by itself. The threshold used is the minimum of a shared array that contains the last dequeued timestamp of each thread.

In addition to our priority queue, we have performed the same experiments on a single ordered linked-list (LList) and on a calendar queue (CQueue), both protected with a single spin-lock. Moreover, we have moved each invocation to the memory allocator outside the critical section imposed by locks. The calendar queue has a maximum number of buckets equal to 32768.

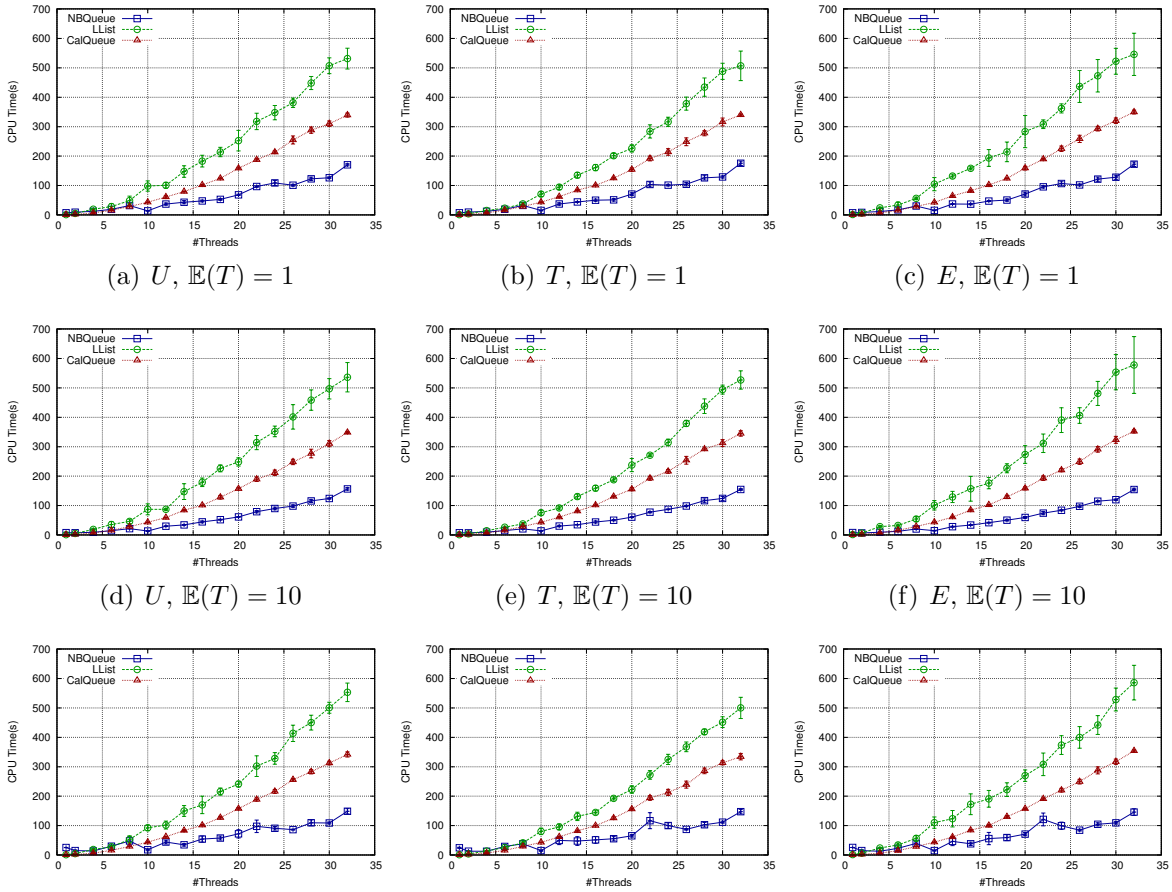
Evaluation Platform The testing platform is a HP ProLiant server equipped with four 2GHz AMD Opteron 6128 processors working at 64 bits. Each processor is an octa-core, giving a total amount of 32 cores. Each core has a private 128 KB L1 cache (64 KB data-cache and 64 KB instruction-cache) and a private 512KB L2 cache. The last level of cache, having 5118 KB capability, is shared among four cores within a single processor, for a total of 10236 KB within the same processor. The machine is equipped with 64 GB of RAM arranged in 8 NUMA nodes of 8 GB. Each NUMA node is close to one group of cores that share the last level of cache and far to the others.

The operating system is 64-bit Debian 8, with Linux Kernel version 2.6.32-5-amd64. The code is written in C and the compiler is gcc 4.9.2 used with no optimization in order to avoid the reordering of memory accesses. The memory allocator used is the standard GNU malloc, ensuring that our results do not benefit from a non-blocking allocator. The random generator used is the GNU `drand48_r` that guarantees thread-safety and a period 2^{48} long.

Metrics The measured metric are the *user time*, *system time* and *real time* obtained from the Linux command *time*. The granularity of the command is 10ms leading to 0.05% of error on the shortest execution. As derived metric we compute the $CPUtime =$*user time*, that is the sum of time that each thread is scheduled in user. Since our hardware was executing only the evaluation program and the operating system, the CPU time captures the fact that in a blocking algorithm, a thread actively waits in order to acquire a resource protected by a spinlock. Finally we compute the average thread throughput as $\frac{\#OPS}{CPUtime}$.

Testing the correctness. In order to verify that our algorithm is correct, we have implemented and tested several sanity checks. In particular we have developed an alternative experiment algorithm, in which instead of stopping the execution when the total amount of operations is completed, we continue to dequeue events until the queue is empty. Then we verify that the number of enqueued items is equal to the dequeued ones.

The second test is the emulation of a parallel event dispatcher in which the thread holding the minimum timestamp is the one allowed to continue the execution. This



(a) U, $\mathbb{E}(T) = 1$ (b) T, $\mathbb{E}(T) = 1$ (c) E, $\mathbb{E}(T) = 1$

(d) U, $\mathbb{E}(T) = 10$ (e) T, $\mathbb{E}(T) = 10$ (f) E, $\mathbb{E}(T) = 10$

(g) U, $\mathbb{E}(T) = 50$ (h) T, $\mathbb{E}(T) = 50$ (i) E, $\mathbb{E}(T) = 50$

Figure 4.3: CPU times for the experiment 1.

ensures that each thread takes from the queue events with timestamps monotonically increasing and always greater than a global minimum.

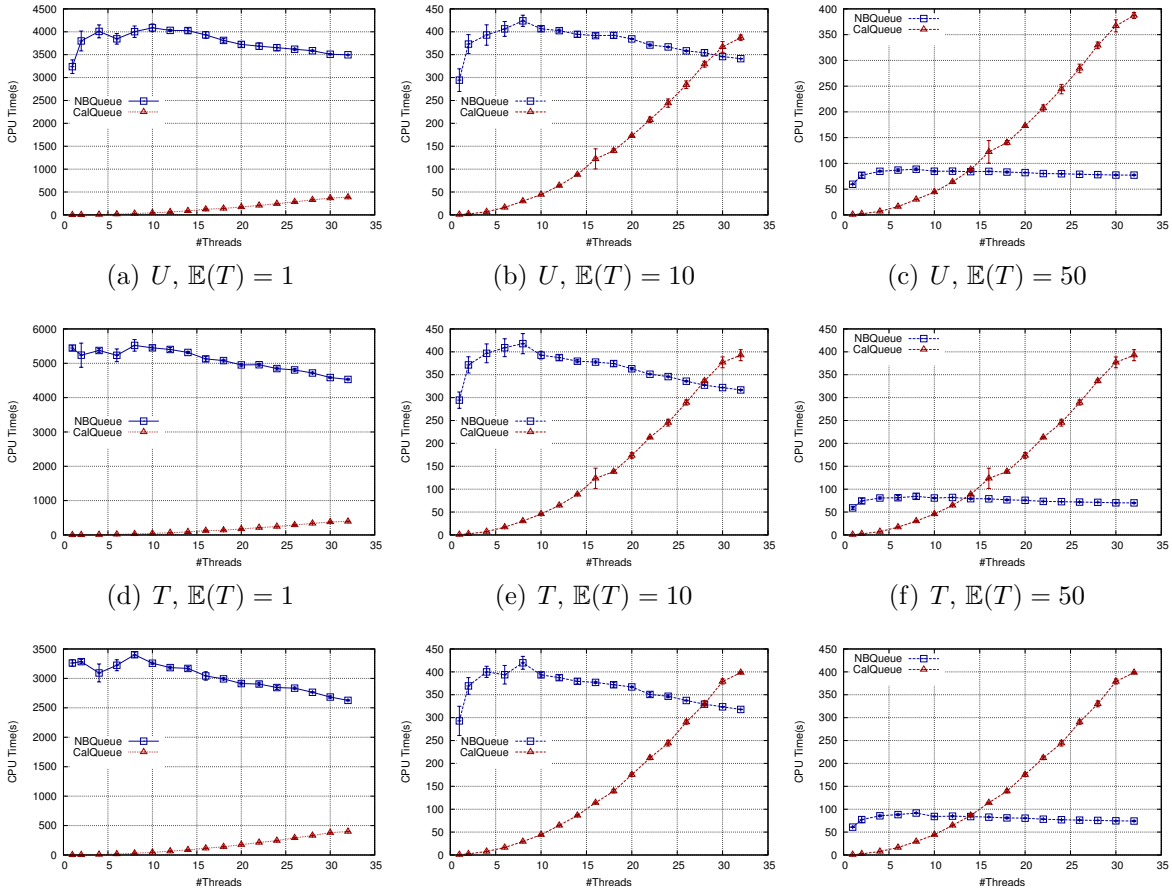
Moreover during every run, PRUNE verifies that there are no events with timestamp lower than the threshold. These checks that every event it is not lost.

Test 1. In this first set of experiments, we have configured an equal probability distribution for ENQUEUE and DEQUEUE routines $P_D = P_E = 0.5$. Although the mean size of the queue should be zero, it is not empty all the time, allowing us to capture the base cost of our algorithm and the effect of an inadequate bucket width.

In Figure 4.3 are shown the *CPUtimes* for all configurations of priority increment distributions and mean values. The linked list and the calendar queue are indifferent to these changes because the queue contains few elements. Thus, the steps performed in the linked list are few, while the calendar queue is more efficient due to its indexed

access that spread events in at least two buckets, reducing the number of items to be traversed per insertion. Our algorithm performs better than alternatives, because ENQUEUEs are fast and non-blocking, while the number of empty bucket is not critical for DEQUEUEs, that are degenerated in incrementing a counter with a `CAS`. In fact, detecting the empty condition is an expensive operation in our solution, because it requires a scan of the whole `table`. This explains why our algorithm have same performances in all the configurations. In order to verify this hypothesis, we have repeated the test with 32 threads and `init_size` equal to 2. This ensures that the `current` index is near to `table` size for most of the time and thus detecting the queue as empty requires fewer buckets to be traversed, obtaining a speed-up from 1.3x to 1.6x. The improvement is limited because we moved the contention from `current` to the Treiber's Stack pointed by `future` or `todo`, which has a similar behavior. A possible solution to detect the queue as empty more efficiently could be the usage of a counter updated with a `fetch&add` instruction, but we have not found a strategy to embed this counter without losing linearizability.

Test 2. The idea is to pre-populate the queue and compare the steady state behavior of the calendar queue against our solution. In [35], Rönngren explains that, for considered distributions, the steady state access time is reached after a number of operations that is five time the queue size. Thus we have imposed $P_D = 0.3$ and $P_E = 0.7$ for the first 30% of the execution, giving an expected queue size about 153600, and $P_D = P_E = 0.5$ for the remaining 760000 operations. In Figure 4.4 it is shown how CQueue performance is indifferent to the used distributions (images from left to right) and their respective mean values (from top to bottom), due to its capability to resize the array and the bucket width. In fact this leads to an access time that is constant with respect to the queue size. On the contrary the linked list has an increasing number of items in a fixed interval, thus its execution time degenerates significantly also in the sequential run and it becomes unreasonable in concurrent execution. Our algorithm degenerates when executed with $\mathbb{E}[T] = 1$ (Figure 4.4(a), 4.4(d) and 4.4(g)), because a single bucket contains half of pending events, making ENQUEUE spend 99.9% of execution time. This result confirms that the capability to variate the bucket width is



(a) U, $\mathbb{E}(T) = 1$ (b) U, $\mathbb{E}(T) = 10$ (c) U, $\mathbb{E}(T) = 50$

(d) T, $\mathbb{E}(T) = 1$ (e) T, $\mathbb{E}(T) = 10$ (f) T, $\mathbb{E}(T) = 50$

(g) E, $\mathbb{E}(T) = 1$ (h) E, $\mathbb{E}(T) = 10$ (i) E, $\mathbb{E}(T) = 50$

Figure 4.4: CPU times for the experiment 2.

crucial when a very huge amount of consecutive insertions occurs in the same bucket. Anyhow a ten times smaller bucket width (Figure 4.4(b), 4.4(e) and 4.4(h)) leads to an execution time that is comparable with the calendar queue, while a fifty times finer grid makes the non-blocking algorithm outperform the rival in all cases (Figure 4.4(c), 4.4(f) and 4.4(i)). It means that increasing the number of buckets is a proper way to exploit the multitude of available core.

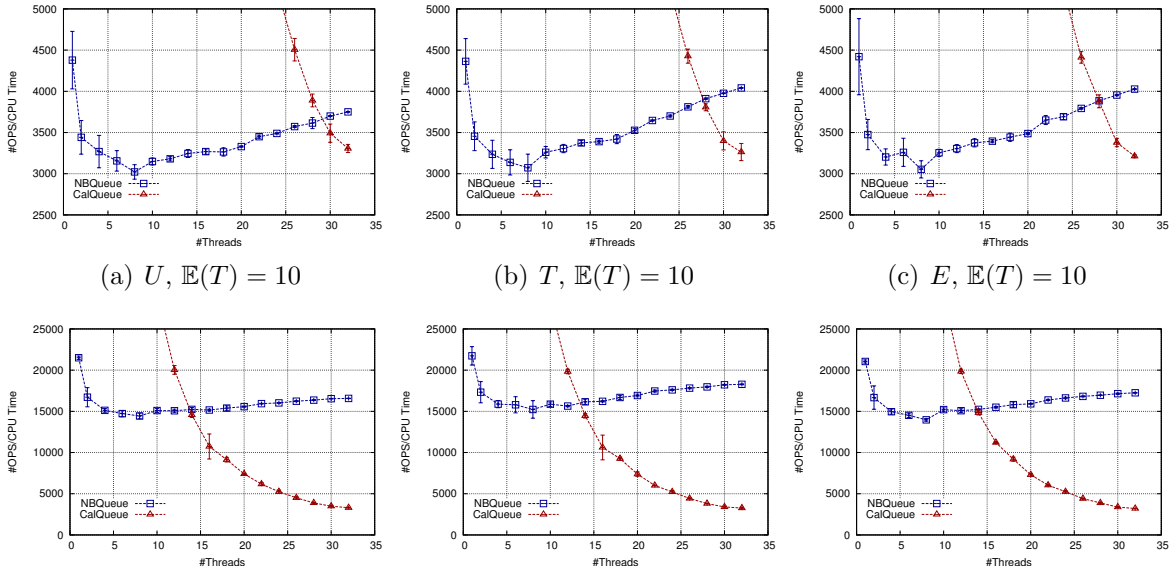
In Figure 4.5 it is shown the average ratio between the bucket width of our solution ($BucketWidth_{NBPQ}$) and the bucket width of the calendar queue at steady state ($BucketWidth_{CQ}$). When this ratio is near to 1, we have chosen a good value for `bucket_width`, since each bucket contains the same amount of events in a bucket of the calendar queue (at least for uniform and triangular distributions).

$\mathbb{E}[T]$	$\frac{BucketWidth_{NBPQ}}{BucketWidth_{CQ}}$
1	≈ 15000
10	≈ 1200
50	≈ 200

Figure 4.5: Relative bucket size of calendar queue.

The difference is that we allocate buckets linearly, while calendar queue uses a circular array. However, when the ratio is about 1:1000, the performance under high contention is comparable. This is an interesting result, because it shows that, in order to have a scalable queue based on calendars and to perform as the blocking algorithm, it is not required to guess an optimal width of the bucket. In fact, we still need that buckets are short enough to guarantee fast ENQUEUEs, but, since we are using non-blocking lists, their length can be about 1000 times longer than blocking alternatives. To confirm that we are handling more events per bucket than the calendar queue, consider the case of uniform distribution with $\mathbb{E}[T] = 10$. We have $BucketWidth_{CQ} \approx \frac{1}{1200}$ and the calendar queue covers an interval of time of size $Y \approx \frac{32768}{1200} \approx 27$. The maximum distance in time between the current minimum and the event with maximum timestamp is 20. It means that all events are contained in $\frac{20}{27} \approx 75\%$ of buckets and the average number of items per bucket is $N_{CQ} \approx \frac{153600}{32768 \cdot 0.75} \approx 6$. Conversely our solution has a bucket width fixed to 1, thus all events are contained in 20 buckets, each one containing $N_{NBPQ} \approx \frac{153600}{20} \approx 7680 \approx N_{CQ} \cdot 1200$ events in average. Similar considerations can be done for triangular and exponential distributions.

Assuming that we have p threads in the system, it should be clear why we can handle a bucket p times longer. In fact when one thread is accessing the calendar queue for a time k, others $p-1$ threads are waiting the same time k. Thus the average access time of each thread is $O(p \cdot k)$. If we want a non-blocking priority queue fast as the blocking calendar queue, we have to spend $O(p \cdot k)$ time per operation, choosing a bucket p times wider than the optimal size. Since we are using $p = 32$ threads, this explain a ratio significantly lower than the measured one. The reason is behind the usage of spin-locks that makes threads in critical section run slower by a factor that is dependent from the contention [36]. Since we have $p-1$ threads that are spinning, the thread holding the lock runs $O(p)$ times slower. It means that a thread completes an operation after $O(p^2 k)$ time. Thus in order to have a comparable execution times with p threads, our algorithm have to run $O(p^2)$ times slower by processing more elements in



(a) U, $\mathbb{E}(T) = 10$ (b) T, $\mathbb{E}(T) = 10$ (c) E, $\mathbb{E}(T) = 10$

(d) U, $\mathbb{E}(T) = 50$ (e) T, $\mathbb{E}(T) = 50$ (f) E, $\mathbb{E}(T) = 50$

Figure 4.6: Thread throughputs for the experiment 2.

the ENQUEUE operation. In other words increasing the sequential work for ENQUEUE is not a dramatical choice provided that we can handle more ENQUEUEs at a time and these are in buckets that do not contain the minimum. DEQUEUEs are indifferent to the bucket width until it is not required a scan to find the next non-empty bucket. In fact a DEQUEUE from a non-empty bucket consists in trying a `CAS` on the first unmarked node, but, in case of unsuccessful exchange, we can restart from the beginning of the bucket. On the contrary the search for a non-empty bucket takes advantage from an increased width, because it allows DEQUEUEs to scan a lower number of buckets. Moreover, even if we are executing fast as the calendar queue, we are in a preferable situation since the throughput of each thread is increasing (Figure 4.6(a), 4.6(b), 4.6(c)), suggesting that we can exploit more processors if available. On the contrary a bucket 200 times larger than the optimum makes the throughput of each thread stable, showing that our algorithm is working to its maximum capability.

Finally, having an increasing number of processors allows us to choose a bucket width distant from the optimum. This simplifies the design of a non-blocking and linearizable algorithm for bucket resize, since it should focused on reducing the number of items per bucket rather than guessing the time distance between two consecutive events.

Chapter 5

Conclusions

In this work we have proposed a non-blocking implementation of a priority queue for Pending Event Set management. Starting from the study of lock-free data structures and of the current state of the art in Pending Event Set problem, we have designed and implemented a priority queue that provides lock-free and constant time accesses for both enqueue and dequeue operations. Our data structure arranges events in a multi-list structure by using lock-free objects, that are well established in the literature. This allowed us to take advantage from their correctness and inherently scalability.

At the best of our knowledge, the proposed solution is the first result that is able to break down the $O(\log n)$ barrier for lock-free priority queues and, at the same time, to ensure linearizability and stability, fundamental proprieties in concurrent environments.

We have also proved the scalability of our approach by implementing the proposed algorithm and testing it with an experimental evaluation on highly parallel hardware. Results have shown that our solution scales well with respect to the number of available cores and can run 5 times faster than a blocking calendar queue under high contention.

We believe that our implementation will be a reference for future works on priority queues, since we have found out lock-free and linearizable solutions to common issues of multi-list approach, that is the basis for each one of the already known priority queues that guarantee constant time accesses.

Bibliography

[1] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[2] Clay Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc., 2009.

[3] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *Solid-State Circuits Society Newsletter, IEEE*, 11(5):33–35, Sept 2006.

[4] H. Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 2005. `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[5] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 176–188, New York, NY, USA, 1991. ACM.

[6] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[7] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[8] Peter Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[9] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec 1966.

[10] Maurice Herlihy and Nir Shavit. On the nature of progress. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, OPODIS'11, pages 313–328, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992.

[12] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[13] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[15] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979.

[16] R. K. Treiber. Systems programming: Coping with parallelism. Technical report, IBM Aladen Research Center, 1986.

[17] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. ACM.

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

[19] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Lock-free dynamically resizable arrays. In *Proceedings of the 10th International Conference on Principles of Distributed Systems*, OPODIS'06, pages 142–156, Berlin, Heidelberg, 2006. Springer-Verlag.

[20] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, December 2007.

[21] J.D. Slingwine P.E. McKenney. Read-copy update: using execution history to solve concurrency problems. In *Proceedings of the 1998 International Conference on Parallel and Distributed Computing and Systems.*, 1998.

[22] Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, 2004.

[23] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

[24] 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, pages 214–222, New York, NY, USA, 1995. ACM.

[25] J.C. Comfort. Simulation of a master-slave event set processor. *Simulation; (United States)*, 3, Mar 1984.

[26] Halim Damerdji and Peter W. Glynn. Performance analysis of future event sets. In *Proceedings of the 27th Conference on Winter Simulation*, WSC '95, pages 316–321, Washington, DC, USA, 1995. IEEE Computer Society.

[27] Jon Bentley. Programming pearls: Thanks, heaps. *Commun. ACM*, 28(3):245–250, March 1985.

[28] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[29] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[30] R. Brown. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *Commun. ACM*, 31(10):1220–1227, October 1988.

[31] Robert Rönngren, Jens Riboe, and Rassul Ayani. Lazy queue: An efficient implementation of the pending-event set. *SIGSIM Simul. Dig.*, 21(3):194–204, April 1991.

[32] Wai Teng Tang, Rick Siow Mong Goh, and Ian Li-Jin Thng. Ladder queue: An o(1) priority queue structure for large-scale discrete event simulation. *ACM Trans. Model. Comput. Simul.*, 15(3):175–204, July 2005.

[33] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627, May 2005.

[34] Sounak Gupta and Philip A. Wilsey. Lock-free pending event set management in time warp. In *Proceedings of the 2Nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '14, pages 15–26, New York, NY, USA, 2014. ACM.

[35] Robert Rönngren and Rassul Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Trans. Model. Comput. Simul.*, 7(2):157–209, April 1997.

[36] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, January 1990.

[37] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, March 1983.

[38] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3):261–322, September 1989.

[39] Brian C. Dean and Zachary H. Jones. Exploring the duality between skip lists and binary search trees. In *Proceedings of the 45th Annual Southeast Regional Conference*, ACM-SE 45, pages 395–399, New York, NY, USA, 2007. ACM.