A Conflict-Resilient Lock-Free Calendar Queue for Scalable Share-Everything PDES Platforms

Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, Francesco Quaglia
Sapienza, University of Rome
{marotta,mianni,pellegrini,quaglia}@dis.uniroma1.it

ABSTRACT

Emerging share-everything Parallel Discrete Event Simulation (PDES) platforms rely on worker threads fully sharing the workload of events to be processed. These platforms require efficient event pool data structures enabling high concurrency of extraction/insertion operations. Non-blocking event pool algorithms are raising as promising solutions for this problem. However, the classical non-blocking paradigm leads concurrent conflicting operations, acting on a same portion of the event pool data structure, to abort and then retry. In this article we present a conflict-resilient non-blocking calendar queue that enables conflicting dequeue operations, concurrently attempting to extract the minimum element, to survive, thus improving the level of scalability of accesses to the hot portion of the data structure—namely the bucket to which the current locality of the events to be processed is bound. We have integrated our solution within an open source share-everything PDES platform and report the results of an experimental analysis of the proposed concurrent data structure compared to some literature solutions.

1. INTRODUCTION

The advent and large diffusion of multi-core machines has impacted the way Parallel Discrete Event Simulation (PDES) platforms are built. They still adhere to the archetypal organization [6] where the overall simulation model is partitioned into interacting distinct entities—the so called simulation objects—yet they do no longer necessarily follow the classical multi-thread organization [27] where objects are (temporarily) bound to specific worker threads. Indeed, the possibility to share any PDES-engine data structure across threads, as well as the actual simulation objects' state, has led to the raise up of new architectural paradigms.

A current trend is the one of building PDES platforms as "share-everything" systems [14]. In this paradigm, an event (which may target any simulation object) can be picked by any worker thread for actual processing. As an extreme, multiple worker threads can even pick events destined to the same simulation object at the same time [19], leading the simulation object to no longer figure out as a sequential entity [17]. Clearly, share-everything PDES platforms allow concentrating the computing power—namely, the worker threads which are operating within the PDES environment—on the higher-priority pending events. Indeed, each worker thread picks events with lower timestamps without considering any partial view of the pending event set—the view generated by partitioning simulation objects across threads. PDES platforms adhering to the share-everything paradigm, or at least tending to this type of architectural organization, can be found in [4, 19, 10, 5, 14].

This re-organization demands for some efficient *global pool* of pending events. Indeed, a core challenge when designing share-everything PDES platforms is to ensure that fully-shared data structures—most notably,the ones used at the level of the simulation engine—guarantee scalability of concurrent accesses. In particular, the pool keeping the pending events destined to whichever simulation object plays a core role to enable scalability of the overall PDES platform. Such a pool should in fact guarantee high concurrency of the accesses for both extraction of the events to be processed, and insertion of newly-scheduled events that result from the processing activities at some object.

Handling concurrent accesses to a fully-shared event pool by relying on critical sections protected via locks (e.g. spin-locks) does not ensure scalability, thus figuring out as an adequate approach limited to a reduced number of threads. For this reason, recent studies [10, 5, 14] have targeted the design of event pool data structures based on the lock-free paradigm. Here concurrent threads attempt to perform their operations—either an extraction or an insertion—without the need to take any lock. To leave the data structure in a consistent state, threads determine whether the operation can be correctly finalized by relying on atomic Read-Write-Modify (RMW) machine instructions [20], offered by off-the-shelf architectures, such as Compare-and-Swap (`CAS`). These instructions are used to determine whether the snapshot of the data structure (or a portion of it) accessed by a thread has been concurrently altered by some other thread. If this occurs, the operation has observed a stale snapshot, and is therefore deemed invalid. Therefore, it is aborted and then retried. In some sense, the operation is seen like an in-memory transaction, which is committed or aborted depending on whether it faces a conflict with a concurrent operation. Although this approach explicitly avoids blocking

SIGSIM-PADS '17, May 24–26, 2017, Singapore.
© 2017 ACM. ISBN 978-1-4503-4489-0/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3064911.3064926

threads, thus enabling resilience versus thread reschedules, it may still be considered sub-optimal in scenarios where the likelihood of conflicts among concurrent operations is non-minimal. In fact, we may still experience a significant volume of operation aborts/retries.

In this article, we take the lock-free calendar queue presented in [14] as our reference and present the design of an innovative version that has the property of being conflict-resilient[1]. Conflict resilience is achieved in relation to concurrent extractions. This is a relevant objective since, in share-everything PDES, concurrent extractions are highly likely bound to the same bucket of the calendar—thus touching the same portion of the whole data structure—namely the bucket containing the event with the minimum timestamp. This bucket is somehow the "hot" one in the whole calendar, since any thread will attempt to pick its next event exactly from that bucket. On the other hand, newly-produced events will more likely carry timestamps associated with different buckets (i.e., buckets in the future), which automatically decreases the likelihood of conflicting on concurrent insertion operations.

As opposed to the approach in [14], the lock-free calendar queue we present in this article makes an extraction operation still valid (i.e., committable) in scenarios where some concurrent extraction has changed the snapshot of the currently hot bucket. Rather, the extraction will be aborted only in case the hot bucket becomes (logically) empty while attempting to extract from it. This is the scenario where a new bucket becomes the hot one, meaning that the locality of the processing activities within the simulation model has moved to a subsequent logical time interval.

We also discuss (and investigate experimentally) how to dynamically resize the width of the buckets in the calendar, which in turn affects the length of the chain of events that are expected to fall within the hot bucket. As we will show, when allowing conflict-resilient lock-free concurrent accesses to the calendar queue, classical policies for resizing the buckets (like the one proposed in the context of sequential accesses to the classical calendar queue presented in [3]) are no longer optimal. This is because the length of the chain of events associated with the buckets affects both:

- the time complexity for the access to the bucket chain, which in our case still stands as amortized constant time; this aspect was already captured by the work in [14];
- the actual number of concurrent extractions from the same bucket which are allowed to be still committable, even if the snapshot of that bucket-chain concurrently changes; this aspect is intrinsically new and specifically related to our conflict-resilient lock-free calendar queue proposal.

The experimental data we report in support of the effectiveness of our proposal, in terms of scalability in face of concurrent accesses, have been collected in differentiated scenarios: one where we test the conflict-resilient lock-free calendar queue as a stand-alone component via synthetic workloads based on the hold model [26], and a second one where the calendar queue has been integrated with a share-everything PDES platform [4], allowing us to test its effects on scalability and performance via both the PHOLD synthetic benchmark for PDES [7] and an agent-based simulation model [12].

The remainder of this article is structured as follows. In Section 2, we discuss related work. The baseline implementation of the lock-free calendar queue we take as the reference for our work is discussed in Section 3. Our conflict-resilient lock-free calendar queue is presented in Section 4. Experimental data are provided in Section 5.

2. RELATED WORK

Event pools constitute core components in both sequential and parallel simulation engines, and the literature on this topic offers a set of differentiated solutions, each one aimed at optimizing the effectiveness of the event pool data structure under specific workloads or execution scenarios. The original Calendar Queue presented in [3] is a timestamp-ordered data structure based on multi-lists, each one associated with a time bucket, which offers amortized constant time insertion of events with generic timestamps and constant time extraction of the event with the minimum timestamp. The Ladder Queue [24] is a variant of the Calendar Queue which is more suited for skewed distributions of the timestamps of the events, thanks to the possibility of dynamically splitting an individual bucket in sub-intervals (i.e., sublists of records) if the number of elements associated with the bucket exceeds a given threshold. The LOCT Queue [16] is an additional variant which allows reducing the actual overhead for constant time insertion/extraction operations thanks to the introduction of a compact hierarchical bitmap indicating the status of any bucket (empty or not). None of these proposals has been devised for concurrent accesses. Therefore, their usage in scenarios with sharing among multiple threads would require a global lock for serializing the accesses, which would be detrimental to scalability, as shown in [8].

The work in [2] provides an event-pool data structure enabling parallel accesses via fine-grain locking of a sub-portion of the data structure upon performing an operation. However, the intrinsic scalability limitations of locking still lead this proposal to be not suited for large levels of parallelism, as also shown in [18].

As for lock-free management of sets by concurrent threads, various proposals exist (e.g., lock-free linked lists [9] or skip-lists [21]), which anyhow do not offer constant time operations. The lock-free linked list pays a linear cost for ordered insertions, while the skip-list pays logarithmic cost for this same type of operation. The proposal in [13] is based on lock-free access to a multi-bucket data structure, and provides amortized $O(1)$ time complexity for both insertion and extraction operations. However, it does not provide a lock-free scheme for the dynamical resize of the bucket width. Hence, to achieve adequate amortizing factors, all the threads would need to (periodically) synchronize to change the bucket width and redistribute events over the reshaped buckets. On the other hand, avoiding at all the synchronized reshuffle of the buckets might give rise to non-competitive amortizing factors (say too many elements associated with a bucket). The proposal in [14] enables non-blocking reshuffle of the bucket width, but does not guarantee conflict resilience of extraction operations targeting the "hot" bucket to which the locality of the activities within the simulation model is bound. Hence as soon as two or more extraction operations are executed concurrently and conflict, just one

[1]The source code of our implementation can be found at https://github.com/HPDCS/CRCQ

of them is allowed to be finalized with no retry cycle. Our present proposal exactly tackles this problem.

Lock-free operations in combination with constant time complexity have also been studied in [8], which presents a variation of the Ladder Queue where the elements are at any time bound to the correct bucket, but the bucket list is not ordered. Constant time is achieved since the extraction from an unordered bucket returns the first available element, which does not necessarily corresponds to the one with the minimum timestamp. This proposal is intrinsically tailored for PDES systems relying on speculative processing, where unordered extractions leading to causal inconsistencies within the simulation model trajectory are reversed (in terms of their effects on the simulation model trajectory) via proper rollback mechanisms. However, still for speculative PDES, a few recent results [4, 19] have shown the relevance of fetching events from the shared pool in correct order, as a means to build efficient synchronization schemes able to exploit alternative forms of reversibility, which stand aside of the traditional Time Warp protocol [11]. Correct order of delivery is guaranteed in our proposal, since we always deliver the highest priority event currently in the event pool, which has been inserted by any operation that is linearized prior to the extraction.

The recent proposal in [10] explores the idea of managing concurrent accesses to a shared pool by relying on Hardware Transactional Memory (HTM) support. Insertions and extractions are performed as HTM-based transactions, hence in non-blocking mode. However, the level of scalability of this approach is limited by the level of parallelism in the underlying HTM-equipped machine, which nowadays is relatively small. Also, HTM-based transactions can abort for several reasons, not necessarily related to conflicting concurrent accesses to a same portion of the data structure. As an example, they can abort because of conflicting accesses to the same cache line by multiple CPU-cores, which might be adverse to PDES models with, e.g., very large event pools. Our proposal does not require special hardware support, thus fully eliminating the secondary effects caused by, e.g., HTM limitations on the abort rate of the operations.

3. RECAP ON THE REFERENCE SOLUTION

As mentioned, our reference solution is the one presented in [14]. The main idea behind this work is to build a lock-free pending event set directly inspired to the classical calendar queue [3]. More in detail, the presented variant is composed of an array of entries referred to as *physical buckets*, each of which is the head element of a lock-free ordered linked list implemented according to [9], with some modifications. The list in [14] relies on the two least-significant bits of the pointer to the next node[2] to represent four different states of a node. The time axis is divided into equal slots, called *virtual buckets*, each of which covers a time interval bw called bucket width. Each virtual bucket is associated with a physical bucket in a circular fashion.

In order to ensure consistency with concurrent accesses, the operations on the lock-free calendar queue rely on atomic RMW instructions, in particular `CAS` and Fetch-and-Add (`Fetch&Add`). The first type of instruction is used to update a given memory location only if its current content is equal to a given value, otherwise no memory update takes place and we say that the `CAS` instruction fails. The second type of instruction gathers the value stored in a given memory location, while atomically incrementing it of a given amount, so any read on that same memory location that is linearized after the execution of `Fetch&Add` necessarily observes the updated value.

When a new event e with timestamp T_e is enqueued, the index I_{pb} of the target physical bucket for the insertion is computed as $I_{pb} = (I_{vb} \mod L)$, where L is the calendar length, namely the number of physical buckets, and I_{vb} is the index of the virtual bucket associated with the time slot TS such that $T_e \in TS$. I_{vb} is computed as $I_{vb} = \lfloor \frac{T_e}{bw} \rfloor$. Finally e is inserted into the I_{pb}-th physical bucket applying the lock-free list-insertion logic defined in [9]. In more detail, the enqueue scans the physical bucket list and when it finds the correct position in the list, it inserts the new event by executing a `CAS` instruction that manipulates the pointer of the node that must precede the one to be inserted. Clearly, given the execution semantic of `CAS`, two concurrent conflicting attempts to manipulate that same pointer will lead one of the operations to fail, thus ensuring consistency of the manipulated list—namely, correct linking of its nodes. The failed insertion operation, if any, is then retried.

An integer C is associated with the calendar, which stores the index of the virtual bucket containing the event with the minimum timestamp, namely the event with the highest priority. A dequeue operation starts by retrieving the value C in order to extract the first node stored into the C-th virtual bucket, which is actually stored into the C_{phy}-th physical bucket, where $C_{phy} = (C \mod L)$. A node is removed by marking it as invalid, flipping the least significant bit of the pointer to the next node with a `CAS` instruction. Clearly, if two concurrent extractions of the minimum try to mark the same node as invalid, then one of them will fail. When the virtual bucket identified by C is empty, the event to extract has to be searched on the next one, then C is incremented by one and the whole operation is restarted.

Going back to enqueue operations, an additional challenge consists in handling insertions in the past of the current minimum timestamp, which may occur either in the current bucket or in a previous one. Although it is known that this type of insertion will never occur for the case of sequential simulation engines, where there is a guarantee of model-wide causality of the generation of the events during model execution, one needs to face this issue when adopting a lock-free calendar queue as the shared pool in the context of PDES systems. As an example, if speculative processing schemes are enabled by the PDES engine, then the current minimum timestamp stored by the calendar queue might not correspond to an event which is causally safe. To cope with such a scenario the proposal in [14] makes the enqueue that inserts an event with timestamp lower than the currently queued minimum repeatedly try to update the value of C with $I_e = \lfloor \frac{T_e}{bw} \rfloor$ by `CAS`, until it has assumed a value $C' \leq I_e$.

Anyhow, this is not sufficient to guarantee correctness. Let us consider an execution where a process P_1 is executing a dequeue and reads a value C_1 of current. After that, P_1 is descheduled and a second process P_2 takes control and executes two consecutive enqueues. The first enqueued event e_a has a timestamp T_a such that $I_a = \lfloor \frac{T_a}{bw} \rfloor$ and $I_a < C_1$. The second event e_b has a timestamp T_b such that $I_b = \lfloor \frac{T_b}{bw} \rfloor$ and $I_b = C_1$. Moreover, let us assume that e_b is now the event

[2]4-bytes alignment ensures that such bits are always set to zero.

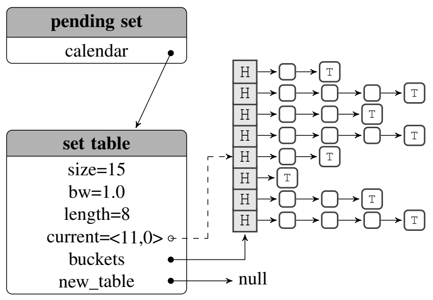


Figure 1: Scheme of the data structure.

with the minimum timestamp in the C_1-th virtual bucket. At this point P_1 is scheduled and continues its operation. In particular it scans the C_1-th bucket and returns e_b. This execution is incorrect since it violates the priority queue semantics, in fact there is a dequeue which returns e_B while e_A was the minimum. For this reason in [14] a dequeue operation re-checks the value of C after it has retrieved the first node from the bucket. If it is unchanged, it means that there has been no enqueue which has been finalized, namely has passed its *linearization point*, and has inserted a node in a previous bucket.

Clearly, any concurrent priority queue has an inherent bottleneck in the retrieval of the element with the minimum timestamp, since at any time it is materialized in a single element, located into a specific bucket [1], while insertion operations can be scattered across different buckets depending on the distribution of the timestamps of the events that are dynamically generated. Given that concurrent dequeues in the approach in [14] try to update the same pointer in the lock-free list via `CAS`, once hit a same node for extraction, just one of them can succeed, leading any other to fully restart. This problem is clearly exacerbated when increasing the level of concurrency of extraction operations. In other words, such concurrent dequeues always try to commit by hitting the first node of the current bucket, thus not attempting to adopt a fall back on the extraction of a subsequent node, without the need to fully restart the dequeue operation. This type of fall back behavior is what we term conflict resilience, which would lead an ongoing extraction operation to slide towards the subsequent element of the bucket list. The complexity of this type of approach stands in how we still allow linearizability of the concurrent operations on the calendar queue, including the possible mix of dequeues and enqueues bound to the same bucket, as we shall discuss.

4. THE CONFLICT-RESILIENT LOCK-FREE CALENDAR QUEUE

The basic organization of the data structure of our conflict resilient calendar queue, which is similar to the one presented in [14], is provided in Figure 1. Essentially it consists of a pointer to a table called *Set Table* which maintains the metadata required for its management. In particular: i) `bw` stores the actual bucket width of the data structure; ii) `length` is the number of buckets in the calendar; iii) `size` keeps the number of stored events; iv) `current` is a pair $\langle index, epoch\rangle$ such that *index* is the index of the virtual bucket containing the minimum and *epoch* is a value whose role will be explained later; v) `buckets` is a pointer to the array that keeps the physical buckets; vi) `new_table` is a reference initialized to `null` which is used only during the resize phase of the calendar. To ensure consistency of concurrent accesses to our data structure, we rely on three different RMW instructions, namely `CAS`, `Fetch&Add` and *Fetch and Or* (`Fetch&Or`).

To avoid the effects of conflicts while concurrently extracting elements from the current bucket, we have introduced a set of capabilities oriented to reduce the need to fully restart the operations, reducing in this way the amount of wasted work. Indeed, reducing the CPU-time required to actually extract a node, by avoiding fully restarting the extraction in case of conflicts, we can reduce the negative performance effects possibly caused by a larger number of conflicting threads.

To achieve this target, we have exploited the newly introduced data-structure field, called *epoch*. As hinted it is stored inside `current` and is used to identify the instant in time at which an operation is performed, in order to reconstruct a partial temporal order among operations. A new epoch starts each time a new node is inserted in the past or in the current virtual bucket. This represents a critical instant in the state trajectory of the calendar queue since it means that a change in the minimum, or in the recorded timestamps that can be close to the minimum, is happened.

When enqueue or dequeue operations are invoked, the ReadTable() procedure is performed to retrieve a pointer to a valid set table. The objective of this procedure is to check if there is a resize operation currently in place on the calendar queue with the goal to assist the threads involved in this operation until it is completed. When no resize operation is in place, the reference to the current valid version of the table can be returned for actually performing extractions or insertions. All the operations on the lock-free calendar queue, including the resize, are described in detail in the next sections.

4.1 Enqueue Operation

The pseudo-code for the Enqueue() operation is shown in Algorithm 1. This operation takes in input an event e and the timestamp T_e associated with it. Once retrieved a valid set table reference using the ReadTable() procedure, the enqueue determines the correct physical bucket associated with the timestamp T_e, computing it as $i = (\lfloor \frac{T_e}{BW} \rfloor \bmod B)$ where BW is the (current) bucket width and B is the size of the calendar, namely the number of physical buckets. Since the i-th bucket is a lock-free linked list implemented according to the indications in [9], we can use the provided search procedure to identify the right point for the insertion of the new node, with minimal changes due to the different states a node can pass through in our implementation. Executing the search procedure we retrieve a couple of nodes, called *left* and *right*, that would surround the new one. When performed, the search procedure tries to compact the nodes marked as deleted that are between the left and the right one, with the aim to return a coherent snapshot of the list. To manage events with the same timestamp, each event stores an increasing sequence number, unique with respect to the nodes with the same timestamp. In this way the couple $\langle timestamp, sequence\_number\rangle$ provides a total order among all stored nodes.

Algorithm 1 lock-free ENQUEUE

```
procedure ENQUEUE(event $e$)
    $tmp \leftarrow$ new node($e$)
    repeat
        $h \leftarrow$ READTABLE( )
        $nc \leftarrow \left\lfloor \frac{n}{h.\texttt{bw}} \right\rfloor$
        $bucket \leftarrow h.\texttt{table}[nc \mod h.\texttt{length}]$
        $\langle left, right \rangle \leftarrow bucket.$SEARCH($tmp.t$, VAL| MOV)
        $tmp.\texttt{next} \leftarrow right$
        $old \leftarrow h.\texttt{current}$
        $tmp.\texttt{epoch} \leftarrow old.\texttt{epoch}$
    until CAS(&$left$.next, UNMARK($right$), $tmp$)
    repeat
        $old \leftarrow h.\texttt{current}$
        if $nc > old.\texttt{value}$ then
            break
    until CAS(&$h$.current, $old$, $\langle nc, old.\texttt{epoch} + 1 \rangle$)
    Fetch&Add(&$h$.size, 1)
```

Algorithm 2 lock-free DEQUEUE

```
procedure DEQUEUE( )
    while true do
        $h \leftarrow$ READTABLE()
        $oldCur \leftarrow h.\texttt{current}$
        $cur \leftarrow oldCur.\texttt{value}$
        $myEpoch \leftarrow oldCur.\texttt{epoch}$
        $bucket \leftarrow h.\texttt{table}[cur++ \mod h.\texttt{t\_size}]$
        $left \leftarrow bucket.\texttt{next}$
        $min\_next \leftarrow left$
        if ISMARKED($left$,MOV) then
            continue
        while $left$.local_epoch $\leq myEpoch$ do
            $right \leftarrow left.\texttt{next}$
            if $\neg$ISMARKED($right$) then
                if $left.\texttt{ts} < cur \cdot h.\texttt{bw}$ then
                    $right \leftarrow$ Fetch&Or(&$left$.next, DEL)
                    if $\neg$ISMARKED($right$) then
                        Fetch&Add(&$h$.size, -1)
                        return $left$.event
                else
                    if $left = \texttt{tail} \wedge h.\texttt{t\_size} = 1$ then
                        return null
                    CAS(&$bucket$.next, $min\_next$, $left$)
                    CAS(&$h$.current, $oldCur$, $\langle cur, myEpoch \rangle$)
                    break
            if ISMARKED($right$,MOV) then
                break
            $left \leftarrow$ UNMARK($right$)
```

Once retrieved the surrounding nodes, the node to be inserted is updated with a reference to the *right* node and with the epoch number retrieved from the `current` field. Finally the node is physically inserted into the list by performing a `CAS` operation on the next field of the *left* node making it point to the new one. If the `CAS` fails, the whole operation starts again from scratch. However, concurrent insertion operations, as we discussed, are typically less critical in terms of conflicts since they likely span different buckets of the calendar, or different surrounding nodes within a bucket.

To complete the update of the structure, the procedure checks if the inserted node belongs to a virtual bucket less than or equal to the one pointed by `current`, namely if the new event stands in the past or around the minimum. In the positive case, it updates the index kept by `current`. As hinted before, the enqueue of a node in the past modifies the minimum element of the queue. Therefore, the epoch number kept by `current` is incremented by one when updating this variable, starting in this way a new epoch of the whole calendar queue. As last operation, the `size` field is incremented by one to reflect the fact that the queue has been increased, in terms of kept elements. Atomicity of all these operations is guaranteed by relying on `CAS` and `Fetch&Add` instructions.

4.2 Dequeue Operation

Compared to the work in [14], the DEQUEUE() operation has been fully reshuffled so that the largest amount of carried-out work is useful also in case of conflicts, thus achieving conflict resilience. The pseudo-code of the DEQUEUE() operation is shown in Algorithm 2. Similarly to the enqueue operation, a dequeue starts by issuing a call to READTABLE(), in order to retrieve a valid table reference. It then fetches the `current` field, in order to extract the *index* of the virtual bucket storing the minimum-timestamp event and the current *epoch*. The physical bucket from which to start the extraction procedure is determined as $i = (index \mod B)$, where B is the size of the calendar. Once the correct physical bucket is identified, it is scanned from the head, looking for a node valid for extraction.

However, we must ensure that the pointed node belongs to the current virtual bucket, since the list associated with a physical bucket might cover multiple virtual buckets. To this end, for each node, we check if the timestamp of the node belongs to the current virtual bucket. Moreover, to avoid the extraction of a node which has just been deleted (unlinked) or marked as invalid (logically deleted), we check whether the two last bits of the next field of the node keep some mark. Finally, according to the new conflict-resilient organization of the queue, we also need to check whether the node is valid with respect to the semantic of the dequeue operation. To this end, we must verify that the node belongs to an epoch earlier than the one read from `current` at the begin of the dequeue operation, or it belongs to the same one. If this condition is true, then the node was inserted before the beginning of the dequeue operation or, alternatively, it was placed in the future (with respect to the current bucket). In this scenario the state of the node is compliant to the semantic of the dequeue operation, thus it can be safely dequeued. In fact, its insertion was correctly linearized before the current (concurrent) extraction. Differently, if the above condition is not verified, then the node was inserted after the beginning of the current dequeue procedure, in particular after the initialization of a new epoch. Therefore, its insertion might have occurred after that a new minimum has been enqueued. In this latter case, the node cannot be extracted.

If the end of the current virtual bucket is reached without finding a valid node, either the current physical bucket is empty or the present nodes belong to a future time slot. In both cases, the procedure must continue the search for the minimum in the next bucket. Hence, `current` is atomically updated using a `CAS`. Regardless of the outcome of the `CAS` machine instruction, the procedure is restarted from the beginning. In this way, if `current` is concurrently updated, the new value becomes visible, even if it identifies a bucket in the past. Moreover, before updating the current bucket, the procedure tries to compact the virtual bucket, by cutting off (unlinking) all the traversed nodes that are found to be marked as invalid.

The search for a valid node halts when it finds a node which respects all the following properties: i) it is not marked, ii) it belongs to a correct epoch, iii) it is associated with the minimum timestamp in the current virtual bucket. Then, the dequeue operation tries to mark it as logically deleted (`DEL`), in order to finally extract and deliver it to the requesting worker thread. Differently from the proposal in [14], the node is marked by relying on `Fetch&Or`. The advantage coming from the reliance on this RMW instruction, rather than the classical `CAS`, is that it successfully completes even if the address of the node is concurrently updated (e.g., due to a concurrent enqueue of a new node). This is clearly favorable to concurrency since we are interested just in marking the node, regardless of a possible next node's update. `Fetch&Or` returns as well the value stored in the target memory location, therefore it is possible to verify the outcome of the marking attempt: if the original value is unmarked, the procedure has successfully marked the node obtaining the event, otherwise the worker thread detects that a concurrent takeover by another thread took place. In this case, the dequeue operation doesn't have to be restarted. Indeed, it is enough to continue the bucket list traversal, since the epoch field can tell whether the enqueue of a node is serialized before or after the current dequeue. Finally, the `size` field is decremented to signal the extraction of an element from the queue.

4.3 Resizing the Queue

$O(1)$ amortized time complexity is guaranteed by the fact that, on average, the number of elements within each bucket is balanced, in a way similar to the original calendar queue. As said before, every time an operation on the queue is performed, a ReadTable() procedure is called to retrieve a reference to a valid table. During this procedure, the number of elements per bucket is checked to determine whether it is still balanced. If this is not the case, a *resize* operation is executed. In particular, the resize is executed if `size` oversteps a certain threshold. The value of this threshold is a function of the desired number of events per bucket (denoted as $DEPB$), and the percentage of non-empty buckets.

The pseudocode of the ReadTable() operation (where the resize operation is implemented) is shown in Algorithm 3. When the *resize* condition is met, to announce its upcoming execution, the `new_table` field of the old set table is pointed to a new (just-allocated) set table. This somehow "freezes" the old table, preventing any new insertion/extraction operation into/from it. From now on, any thread executing a ReadTable() operation will be aware that a resize operation is taking place, and will start to participate.

Once the reference to the new table is published (preventing any thread from using the old one), before starting to migrate the nodes from the old to the new table, we must mark as `MOV` every entry of the bucket array, and all the first valid nodes of the associated lock-free lists. This is necessary to abort the execution of any dequeue operation. In fact, dequeue operations are restarted any time a node marked as `MOV` is found. In particular, a dequeue operation is restarted (and possibly joins the queue resize operation) if it attempts to dequeue any first valid node, or it tries to insert a new node to a list's head.

Algorithm 3 lock-free ReadTable

```
1:  procedure ReadTable()
2:      $h \leftarrow$ array
3:      $curSize \leftarrow h$.size
4:      if $h$.new = null $\wedge$ resize is NOT required then
5:          return $h$
6:      compute $newSize$
7:      CAS(&$h$.new, null, new array($newSize$))
8:      $newH \leftarrow h$.new
9:      if $newH$.bw$\leq 0$ then
10:         $begin \leftarrow$ RANDOM()
11:         for $j \leftarrow 0$ to $h$.t_size$-1$ do
12:             $i \leftarrow (begin + j)$mod$curSize$
13:             retry-loop to mark $i$-th head as MOV
14:             retry-loop to mark first node of $i$-th bucket as MOV
15:         $MST \leftarrow$ compute bucket width
16:         CAS(&$newH$.bw, $-1.0$, $MST$)
17:     for $i \leftarrow 0$ to $h$.length$-1$ do
18:         while $j$-th bucket of $h$ is non-empty do
19:             $i \leftarrow (begin + j)$mod$curSize$
20:             get first node of bucket $i$ as $right$
21:             get the next node of $right$ as $right\_next$
22:             if $right$ = tail then
23:                 break
24:             if $right\_next \neq$ tail then
25:                 retry-loop to mark it as MOV
26:             create a $copy$ of the $right$ node
27:             while $true$ do
28:                 search for $right.ts$ in a virtual bucket $vb$ of $newH$
29:                 if found node $n$ with same key then
30:                     release $copy$
31:                     $copy \leftarrow n$
32:                     break
33:                 else if successful to insert $copy$ as INV with a CAS then
34:                     break
35:             if CAS(&$right$.replica, null, $copy$) then
36:                 Fetch&Add(&$newH$.size, 1)
37:             else if $right$.replica$\neq copy$ then
38:                 try-loop to mark $copy$ as DEL
39:             retry-loop to ensure that $newH$.current.value $\leq vb$
40:             retry-loop to mark $right$.replica as VAL
41:             retry-loop to mark $right$ as DEL
42:     CAS(&q.array, $h$, $newH$)
43:     return $newH$
```

We are therefore sure that no one will extract nodes from the queue, making stable the portion of the time axis close to the minimum. This assumption allows us to safely determine the new bucket width, and the length of the bucket array—how to determine the bucket width will be discussed in Section 4.4. To this end, in a way similar to [3], a certain amount of events (starting from the minimum) is inspected to compute the *mean timestamp separation*. This is the average distance between the timestamps of consecutive events along the time axis. This value is multiplied by the desired number of events per bucket, in order to compute the bucket width. The result is stored in the `bw` field of `new_table`, by relying on a single-shot `CAS` (i.e., the operation is not retried if it fails).

In order to ensure lock-freedom during the resize operation (without introducing multiple copies of the same node), we rely on a mark-and-clone strategy. Before migrating a node, a thread must ensure that the node itself and its successor are marked as `MOV`. If they are not, it tries to do so by relying on CAS. This is done in order to avoid any interference with concurrent threads, executing any enqueue/dequeue operation, which did not notice that a resize operation is taking place. If the CAS succeeds (or if the nodes are already marked as MOV), the thread allocates a copy of the node, this time marked as invalid (INV). This copy is placed in the new structure, and the node (marked as INV) is ignored by any other dequeue operation until it is validated. By using a copy of the node, it is guaranteed that no node is lost. When placing the node's copy into new_table, if a copy of the same node is already present, it means that some other thread has already performed this operation. In this case, the copy is released and a reference to the one already installed into new_table is returned. Then, the node in the old structure is atomically updated so as to keep a reference to the found new copy. By traversing this reference, the node's copy in new_table is transitioned to the valid state VAL, and the original node is marked as logically deleted (DEL). It will be later physically deleted (i.e., unlinked from the list), still using CAS. To understand the reason behind this protocol, we should consider the insertion of a copy of one node performed by a delayed thread. In this scenario, the resize operation might be already finished, and a copy of the node could be already placed in new_table. Thanks to the validation of the new copy, realized by publishing a pointer to it in the original node, we are sure that the new copy will be not validated. In fact, until the original node is not removed, it will reference its copy. This copy-based pattern allows to enforce lock-freedom. In fact, any thread can take on the job of concurrently migrating the same node, trying to finally flag the original node and its new copy.

4.4 Varying the Bucket Width

In the original calendar queue [3], a strategy is defined to compute the bucket width, which estimates the mean separation time (MST) between two consecutive events. The obtained MST is then multiplied by a constant equal to 3, which represents the desired number of events per bucket (denoted as $DEPB$). This guarantees that the number of events in a bucket is bounded by a constant and thus the calendar queue delivers O(1) amortized time complexity for its operations. The value of $DEPB$ is chosen according to an experimental evaluation performed by Brown, which shows that such a value delivers good performance under different distributions of the timestamps.

Anyhow, the original calendar queue might suffer from reduced performance in at least two pathological scenarios. The first one occurs when the queue has reached a steady size, but the priority increment distribution changes over time, leading to a different MST compared to the one measured during the last resize. It follows that the bucket width is inappropriate and performance deteriorates. In the second case, the priority increment distribution makes events be clustered into two buckets distant from each other and reduces the efficiency, since enqueues traverse a significant amount of events during insertions and dequeues scan a large number of empty buckets.

There are several works that try to resolve these issues by triggering the data-structure reshuffle more frequently and designing new strategies to individuate a more accurate bucket width. In particular, the authors in [15] state that the sampling of events should be obtained from the most dense buckets, namely, those that contain the highest percentage of events stored into the queue. An analytical model is presented in [23] for computing a scale factor for the actual bucket width such that the new bucket width minimizes (in the model) the cost of queue operations. The final bucket width computation resorts to a combination of the sampling technique described in [15] and a minimization technique. Moreover the condition triggering the data-structure reshuffle is checked periodically. The time period is selected accurately in order to maintain $O(1)$ amortized access time. The authors in [24] define a new data structure able to recursively split individual and dense buckets, making the bucket width be chosen properly for each bucket.

Anyhow, these works define heuristics and algorithms for computing MST, $DEPB$ and the bucket width for the case of sequential accesses. So they do not account for the effectiveness of the proposals with concurrent, scalable, conflict-resilient multi-list event pools, like the one we are presenting. They try to minimize the average cost of queue operations by modeling properties of events, in particular MST, so they cannot capture dynamics and behaviors connected to interactions between concurrent threads. In order to cope with such aspects, so as to determine a suited bucket width for our concurrent event pool, we had to take into account the impact of retries on the cost of queue operations, which lead to repeat some computational steps, such as atomic instructions. In particular, when a dequeue finds an empty bucket, it tries to increase current with a CAS instruction. Updating such variable is an expensive operation, since it is likely contended among threads, and might at worst lead to thrashing behaviors, just like for the Treiber Stack [25]. Consequently, it is reasonable that the number of events in a bucket should be at least T_d in our approach, where T_d is the number of cores expected to concurrently access a bucket for dequeue operations, which can be clearly less than the total number of cores available for running the share everything PDES system, which we denote as n. Anyhow, T_d can be much greater than the value of $DEPB$ used to compute the bucket width according to the rule proposed with, e.g., the original calendar queue. On one hand, this should not affect the asymptotic cost of dequeue operations, since a wider bucket width decreases the probability to scan a large amount of empty buckets. On the other hand, longer bucket lists affect the enqueue access time, since the enqueue has to scan an increased number of events before finding the position for the insertion. However, as shown in [13], a highly concurrent event pool based on a multi-list approach can have $O(n^2)$ times longer bucket lists than a baseline solution relying on spinlocks, and obtain comparable performance under high concurrency levels. Therefore, it follows that an average number $DEPB$ of elements within a bucket such that $T_d < DEPB << n^2$ can be a desirable value for optimizing the cost of both enqueue and dequeue operations in our proposal. Given that $T_d \leq n$ independently of the actual access pattern to the queue by concurrent threads, we suggest a value $DEPB \approx n$ as a suited one. In any case, in the experimental section, we report data with different configurations of the parameter $DEPB$, just to capture the effects of possibly different parameterizations while determining the bucket width.

5. EXPERIMENTAL DATA

We experimentally evaluated the performance of our data structure with two different test settings. In the first one

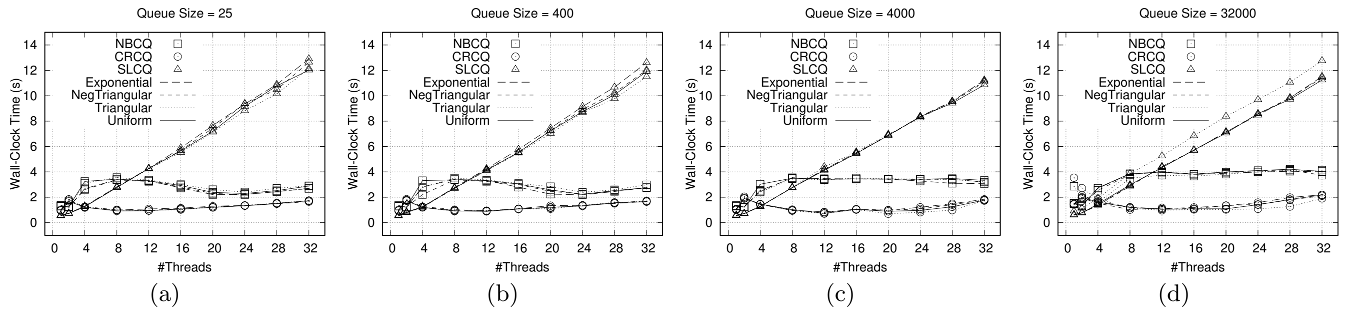


Figure 2: Results with the Hold-Model (each plot refers to a different queue size).

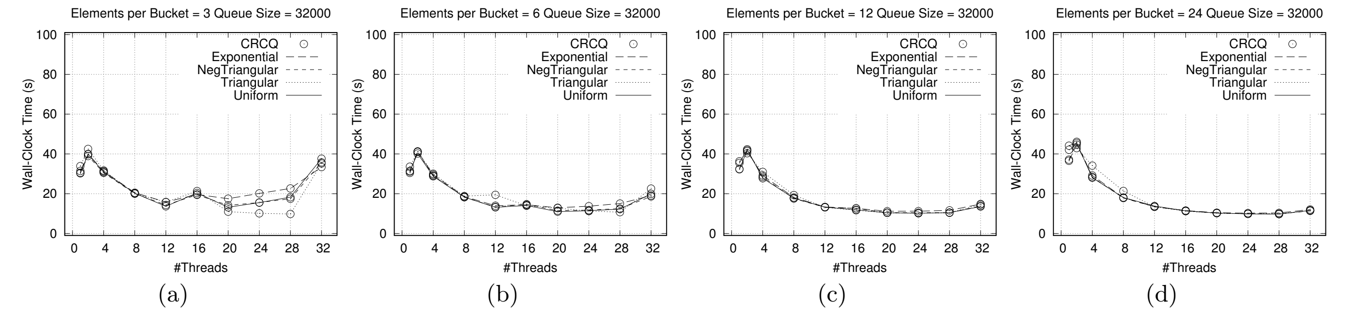


Figure 3: Hold-Model wall-clock times with queue size equal to 32000 and varied $DEPB$.

we exploited a synthetic workload based on the well known *Hold-Model*, where *hold* operations, namely a dequeue followed by an enqueue, are continuously performed on top the queue, which is pre-populated with a given number of items at startup (referred to as queue size). This test allows evaluating the steady-state behavior of the queue and, when executed in a multi-threaded fashion, its scalability and resilience to performance degradation in scenarios with scaled up volumes of concurrent accesses. The second test settings is related to the integration of the presented lock-free queue within an open source share-everything PDES environment, on top of which we run both the classical PHOLD benchmark for PDES systems and a multi-robot exploration model, called TCAR. The platform used in all the experiments is a 32-core HP ProLiant machine running Linux (kernel 2.6) equipped with 64 GB of RAM. The number of threads running the test-bed programs has been varied from 1 to 32.

5.1 Results with the Hold-Model

With the Hold-Model workload, the event pool is gradually populated with a given number of elements (queue size) and then each concurrent thread performs dequeue/enqueue operations with equal probability set to 0.5. Each run terminates when the total number of operations executed by the threads reaches 10^6. We run experiments with 4 different distributions of the priority increment for the generation of new elements to be inserted into the queue, which are shown in Table 1. For each distribution, we executed 4 tests with varies queue size, say 25, 400, 4000 and 32000. The performance of the proposed conflict-resilient calendar queue (CRCQ) is compared with the classical Calender Queue [3] (SLCQ), whose concurrent accesses are synchronized via spin-locking, and the lock-free $O(1)$ event pool presented in [14] (NBCQ).

The results are shown in Figure 2, where each sample is obtained as the average of 10 different runs of the same configuration. We report the wall-clock times required to perform the target number of operations while varying the number of threads from 1 to 32. As expected the evaluated data structures have $O(1)$ access time, in fact, once the number of running threads is fixed, the wall-clock time is constant for different combinations of queue sizes and priority increment distributions. Both the two tested lock-free solutions outperform SLCQ. However, our new proposal CRCQ shows an improved performance wrt the NBCQ in almost every scenario, just thanks to conflict resilience on dequeues. Moreover, any number of threads larger than 4 is enough to make CRCQ be the most efficient implementation among all the tested alternatives.

Although the wall-clock time shown by CRCQ slightly increases with more than 12 threads, it is reasonable to think that the flat line of NBCQ in Figures 2(c) and 2(d) does not reveal a behavior similar to the ones in Figures 2(a) and 2(b) just because the minimum wall-clock time is expected with a higher number of threads compared to the maximum we could experiment with[3]. In other words, the higher efficiency of CRCQ allows to sense earlier the inherent bottleneck represented by retrieving the minimum from the event pool since, with reduced queue size, each bucket

[3]In fact, experimenting with a number of threads greater than the number of available CPU-cores in the underlying platform would have generated scenarios of interference on CPU usage, possibly leading to unclear motivations for winning or loosing on the comparison among the different solutions.

Table 1: Employed Timestamp Increment Distributions.

Probability Distribution	Formula
Uniform	$2 \cdot \texttt{rand}$
Triangular	$\frac{3}{2} \cdot \sqrt{\texttt{rand}}$
Negative Triangular	$3 \cdot (1 - \sqrt{\texttt{rand}})$
Exponential	$-\ln(\texttt{rand})$

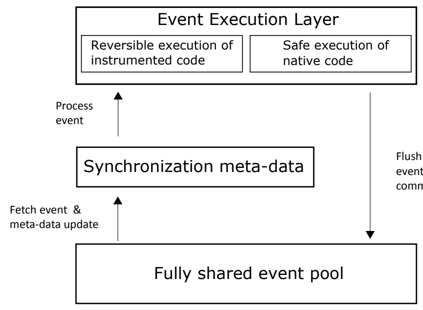

Event Execution Layer
Reversible execution of instrumented code
Safe execution of native code
Process event
Synchronization meta-data
Flush new events upon commit
Fetch event & meta-data update
Fully shared event pool

Figure 4: High level structure of the employed share-everything PDES platform.

contains few items and consequently the probability that a dequeue finds an empty bucket is higher.

To confirm this hypothesis, we have run a synthetic test with a queue size equal to 32000, but this time with different values of $DEPB$, namely 3, 6, 12, 24. As shown in Figure 3, an increased number of items in a bucket allows to increase the performance improvement obtained by a scaled up number of threads with CRCQ, since this reduces the probability to find an empty bucket. Consequently the amount of "wasted" local work of a thread is also reduced, thanks to conflict resilience of the proposed data structure, and at the same time there are less conflicts in updating/reading the value of `current`. Moreover, longer buckets act as an implicit back-off mechanism, exalting the resilience capability of our proposal to conflicting concurrent accesses to the hot spot of the queue, namely the minimum timestamp event. We recall again this is materialized in only one item at any time. This also confirms the deductions in [13] related to the fact that the optimal bucket width in scenarios with concurrent accesses can be significantly different from the one characterizing sequential implementations.

5.2 Share-everything PDES Results

In order to test our proposal in a pragmatic PDES scenario, we have integrated it with the last generation share-everything PDES engine standing at the core of the RAMSES speculative simulation framework [4]. The basic architecture of this engine is shown in Figure 4.

A meta-data layer is used to track the simulation object currently being run by any thread and the timestamp of the corresponding event. Thanks to the share-everything nature of the platform, there is only one event pool shared among all threads. Each extraction from the underlying event pool by some worker thread entails a meta-data update, which allows to compute the *commit horizon* of the simulation and to distinguish if an event can be affected by other events in the past or not. In the first case, the extracted event can be safely executed by triggering the native version of the application code. Otherwise, the worker thread speculatively processes the event by running a modified version of the event handler obtained thanks to a transparent instrumentation (an ad-hoc compile/link procedure) of the native code. The instrumented code generates at runtime undo blocks of machine instructions, which can be used to rollback updates performed by the event processing. Moreover the simulation engine exploits lookahead in order to pinpoint events (after the commit horizon) that will not eventually be affected by causality errors and thus can be executed safely. The events produced by a speculative execution are locally stored waiting for the commitment of the event that generated them. If that event is not eventually committed because of a causality error, the local buffer of events is simply discarded without affecting the pending event set. Consequently, the event pool keeps only the *schedule committed* events, namely those generated by events that will never be rolled back. The concurrent extraction of events targeting the same simulation object leads to conflicting accesses by multiple threads to the state of that same object. These are resolved by synchronizing the worker threads via read/write spin-locks (giving higher priority to lower timestamp events). The original implementation of this simulation engine has been based on a shared event pool implemented as a calendar queue protected via spin-locks. Thus, resorting to the proposed conflict-resilient, lock-free and constant-time event pool has the advantage of confining the explicit synchronization across threads via spin-locks (e.g. in case of concurrent accesses to the same object) outside the lowermost layer. This is expected to provide significantly enhanced scalability.

5.2.1 Results with PHOLD

The first test-bed application we run on top of the share-everything engine is the classical PHOLD benchmark [7]. We configured it with 1024 simulation objects. The execution of an event leads to update the state of the target simulation object, in particular statistics related to the advancement of the simulation, such as the number of processed events and average values of the time advancement experienced by simulation objects. It also leads to executing a classical CPU busy-loop for the emulation of a given event granularity. There are two types of events: i) *regular* events, whose processing generates new events of any type; ii) *diffusion* events that do not generate new events when being processed. The number of diffusion events generated by regular ones (denoted as Fan-Out) is set to 1 and 50 in our evaluation. This varied event pattern leads to scenarios where the average number of events in the event pool is stable, but there are punctual fluctuations, which are more or less intense. In turn, these can allow assessing the effects of our proposal in scenarios where the actual locality of the activity (in particular extraction activities) bound to the hot bucket of the calendar queue can be more or less intense.

The timestamp increments are chosen according to an exponential distribution with mean set to one simulation-time unit. We selected two different lookahead values, respectively 10% and 0.1% of the average timestamp increment, in order to observe the impact of more safe vs more speculative processing on the run-time dynamics. Finally, the busy loop while processing an event is set to generate different event granularity in different tests, namely $60\mu s$, $40\mu s$ and $22\mu s$, in order to emulate low to medium granularity events proper of a large variety of discrete event models.

In Figure 5 and Figure 6 we show the speedup achieved by the parallel runs wrt the sequential execution of the same model. Speedup values are shown while varying the number of threads in the share-everything PDES system from 1 to 32, and with different lookahead values (respectively 10% and 0.1% of the average timestamp increment). Whenever the event granularity is larger and the lookahead is large enough to make threads process events safely with high probability (Figure 5(c)), we observe that both the lock-free event pools allow an almost linear speedup with coefficient 1 (ideal speedup). This is observed up to 16 threads. When running with more threads, the speedup coefficient is 0.8 for CRCQ and 0.7 for NBCQ. Conversely, the spin-locked calendar queue has scalability problems with more than 8 threads and its efficiency is further negatively affected by an increased Fan-Out value. Reduced lookahead (Figure6(c)) does not affect significantly the behavior of lock-free solutions and makes the spin-lock protected calendar queue to achieve an increased speedup when the Fan-Out is set to 1. The reason behind this behavior is that a smaller lookahead increases the probability to extract an unsafe event, that has to be executed speculatively and requires an explicit synchronization (possibly with rollback) in the worst case, reducing the actual concurrency in accessing the shared event pool.

A reduced event granularity leads to a higher pressure on the shared event pool by increasing concurrent accesses. In particular an event granularity equal to $40\mu s$ and a lookahead set to 10% (Figure 5(b)) makes the spin-locked calendar queue deteriorate after 8 threads and NBPQ after 24 threads (mostly because of conflicts and retries on dequeue operations), while CRCQ still delivers linear speedup. A reduced lookahead (Figure 6(b)) leaves almost unaltered the speedup of both the lock-free queues, but leads to a trend reversal. In particular, a larger Fan-Out makes lock-free solutions achieve higher performance (conversely the spin-locked calendar goes worse) since it balances the trend of reduction of concurrency in accessing the shared event pool due to synchronization at the meta-data layer (thanks to more intense bursts of enqueue operations). Anyhow also in this adverse scenario the proposed solution still delivers an almost linear speedup.

Finally, a very fine grain event, namely $22\mu s$, leads to a scenario quite similar to the tests with the Hold-Model, where increasing the number of threads leaves unaltered the wall-clock time spent into the conflict-resilient calendar queue. In Figure 5(a) it is shown how the speedup slope is reduced when moving from 24 to 32 threads in our solution (anyhow the speedup is still 0.5 of the ideal one with 32 threads), while a smaller lookahead (see Figure 6(a)) makes our proposal still achieve the same speedup of larger lookahead scenarios when Fan-Out set to 50, and gives no speedup advantages when moving from 24 to 32 number of threads with the smallest Fan-Out, thus indicating how CRCQ does not negatively impact the absolute performance when moving towards the usage of the maximum admitted parallelism level.

5.2.2 *Results with TCAR*

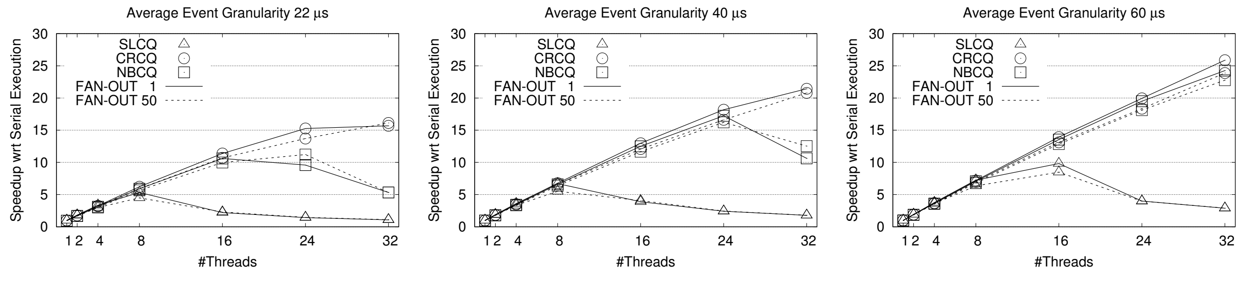
As the second PDES test-bed, we used a variant of the Terrain-Covering Ant Robots (TCAR) model presented in [12]. In this model, multiple robots (say agents) are located into a region (the terrain) in order to fully explore it. TCAR simulations are usually exploited to determine tradeoffs between the number of employed robots, and the latency for exploring the target region, e.g., for rescue purposes. Factors such as the speed of movement (depending on, e.g., environmental conditions within the region, or even obstacles) can be also considered.

In our implementation of TCAR, the terrain to be explored is represented as an undirected graph, therefore a robot is able to move from one space region to another in both directions. This mapping is created by imposing a specific grid on the space region. The robots are then required to visit the entire space (i.e., cover the whole graph) by visiting each cell (i.e., graph node) once or multiple times. Differently from the original model in [12], we have used hexagonal cells, rather than squared ones. This allows for a better representation of the robots' mobility featuring real world scenarios since real ant robots (e.g., as physically realized in [22]) have the ability to steer to any direction.

The TCAR model relies on a node-counting algorithm, where each cell is assigned a counter that gets incremented whenever any robot visits it. So the counter tracks the number of *pheromones* left by ants, to notify other ones of their transit. Whenever a robot reaches a cell, it increments the counter and determines its new destination. Choosing a destination is a very important factor to efficiently cover the whole region, and to support this choice the trail counter is used. In particular, a greedy approach is used such that, when a robot is in a particular cell, it targets the neighbor with the minimum trail count. A random choice takes place if multiple cells have the same (minimum) trail count.

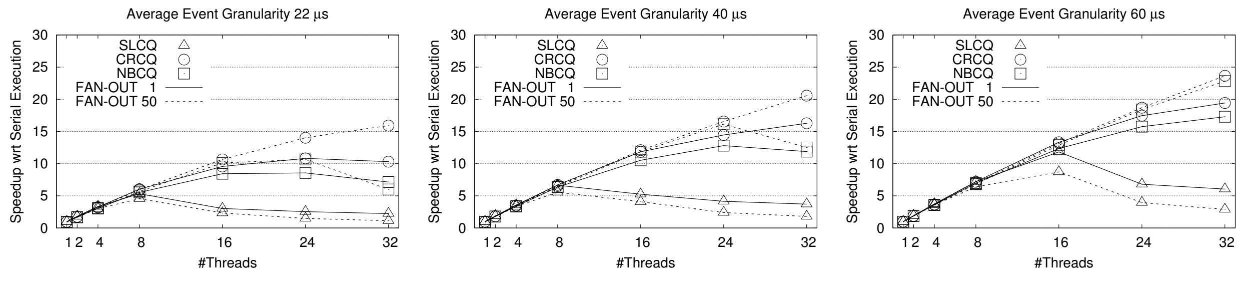
The original TCAR model adopts a *pull* approach for gathering trail counters from adjacent cells. Considering the traditional PDES programming model, based on data separation across the simulation objects, such an approach would result in sending query/reply events across objects modeling adjacent cells each time a robot needs to move to some new destination cell.

To reduce the interactions across the simulation objects (by reducing the volume of events to be scheduled along the model lifetime) we adopted a *push* approach, relying on a notification event (message) which is used to inform all neighbors of the newly updated trail counter whenever a robot enters a cell. Then, each simulation object modeling a cell stores in its own simulation state the neighbors' trail-counters values, making them available to compute the destination when simulating the transit of a robot. In the used TCAR configuration, we included the evaluation of a new state value for the cell whenever a robots enters it, so as to mimic the evolution of a given phenomenon within the cells. This computation has been based on a linear combination of exponential functions (like it occurs for example when evaluating fading on wireless communication systems due to environmental conditions). Further, to model the delay robots experience when entering a cell for correctly aligning itself spatially, a lookahead of 10% of the average cell residence time has been added when generating new events used to notify the update of the local trail counter to the neighbors. We configured TCAR with 1024 cells, and we studied two alternative scenarios with different ratios between the number of robots exploring the terrain and the number of



(a) (b) (c)

Figure 5: Results with PHOLD and lookahead 10% of the average timestamp increment.



(a) (b) (c)

Figure 6: Results with PHOLD and lookahead 0.1% of the average timestamp increment.

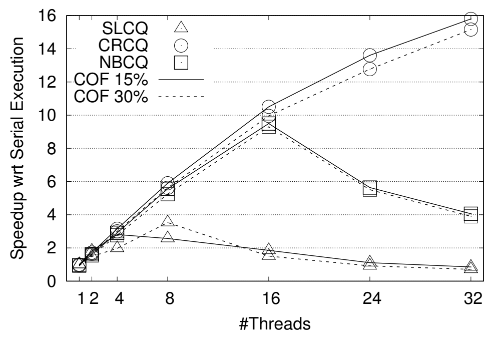


Figure 7: Results with TCAR.

cells, say 15% and 30%. We refer to this parameter as Cell Occupancy Factor (COF).

Speedup results are shown in Figure 7, where we still keep the PDES engine configuration with spin-lock protected calendar queue as the reference. By the plot we see how, also for this application, the maximum thread count leading the spin-lock protected calendar queue to be competitive is between 4 and 8, depending on the value of COF. Beyond 8 threads, such configuration rapidly degrades. NBCQ scales well up to 16 threads, but then degrades, while CRCQ provides close to linear speedup up to 32 threads, jointly guaranteeing at least 50% of the ideal speedup value even for larger thread counts.

6. CONCLUSIONS

In this article we have presented a conflict-resilient non-blocking calendar queue suited for share-everything PDES systems. Our proposal allows multiple worker threads that attempt to extract the element with the minimum timestamp from a fully shared event pool to survive in their operations, despite the conflicting access to (and attempt to modification of) the same portion of the data structure—namely the hot bucket containing the unprocessed events with lower timestamps. With our solution, we experimentally determined excellent scalability of the shared-everything PDES engine on a machine with 32 CPU-cores, thus running the PDES applications with up 32 concurrent worker threads. We also tested our proposal with workloads related to concurrent accesses aside of the one generated by thread operating within the share-everything PDES platform. Besides the achievement of such scalability levels, our proposal also opens a new way of looking at non-blocking data structures, since it leaves the classical path where a non-blocking concurrent data structure is implemented by having a conflicting operations to be necessarily aborted and then retried from scratch.

7. REFERENCES

[1] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The SprayList: a scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 11–20, 2015.

[2] R. Ayani. LR-Algorithm: concurrent operations on priority queues. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, SPDP, pages 22–25, 1990. IEEE Computer Society.

[3] R. Brown. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[4] D. Cingolani, A. Pellegrini, and F. Quaglia. RAMSES: Reversibility-based agent modeling and simulation

environment with speculation support. In *Proceedings of Euro-Par 2015: Parallel Processing Workshops*, PADABS, pages 466–478. LNCS, Springer-Verlag, 2015.

[5] T. Dickman, S. Gupta, and P. A. Wilsey. Event pool structures for PDES on many-core Beowulf clusters. In *Proceedings of the 2013 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 103–114. ACM Press, 2013.

[6] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[7] R. M. Fujimoto. Performance of Time Warp Under Synthetic Workloads. In *Proceedings of the Multiconf. on Distributed Simulation*, pages 23–28. Society for Computer Simulation, 1990.

[8] S. Gupta and P. A. Wilsey. Lock-free pending event set management in time warp. In *Proceedings of the 2014 ACM/SIGSIM Conference on Principles of advanced discrete simulation*, PADS, pages 15–26. ACM Press, 2014.

[9] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC, pages 300–314. Springer Berlin/Heidelberg, 2001.

[10] J. Hay and P. A. Wilsey. Experiments with hardware-based transactional memory in parallel simulation. In *Proceedings of the 2015 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, pages 75–86. ACM Press, 2015.

[11] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and System*, 7(3):404–425, 1985.

[12] S. Koenig and Y. Liu. Terrain coverage with ant robots: a simulation study. In *Proceedings of the fifth international Conference on Autonomous agents*, AGENTS, pages 600–607. ACM Press, 2001.

[13] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia. A Non-Blocking Priority Queue for the Pending Event Set. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, SIMUTools, pages 46-55.ICST, 2016.

[14] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia. A lock-free O(1) event pool and its application to share-everything PDES platforms. In *Proceedings of the 20th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, DS-RT, pages 53–60, 2016.

[15] S. Oh and J. Ahn. Dynamic calendar queue. In *Proceedings 32nd Annual Simulation Symposium*, pages 20–25, 1999.

[16] F. Quaglia. A low-overhead constant-time lowest-timestamp-first CPU scheduler for high-performance optimistic simulation platforms. *Simulation Modelling Practice and Theory*, 53:103–122, 2015.

[17] F. Quaglia and R. Baldoni. Exploiting Intra-Object Dependencies in Parallel Simulation. *Inf. Process. Lett.*, 70(3):119–125, 1999.

[18] R. Rönngren and R. Ayani. A comparative study of parallel and sequential priority queue algorithms. *Transactions on Modeling and Computer Simulation*, 7(2):157–209, 1997.

[19] E. Santini, M. Ianni, A. Pellegrini, and F. Quaglia. Hardware-transactional-memory based speculative parallel discrete event simulation of very fine grain models. In *2015 IEEE 22nd International Conference on High Performance Computing*, HiPC, pages 145–154. IEEE Computer Society, 2015.

[20] D. J. Sorin, M. D. Hill, and D. A. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[21] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627. Academic Press, Inc., 2005.

[22] J. Svennebring and S. Koenig. Building Terrain-Covering Ant Robots: A Feasibility Study. *Autonomous Robots*, 16(3):313–332, 2004.

[23] K. L. Tan and L.-J. Thng. Snoopy calendar queue. In *Proceedings of the 32nd Conference on Winter simulation*, WSC, pages 487–495, 2000.

[24] W. T. Tang, R. S. M. Goh, and I. L. Thng. Ladder queue: An O(1) priority queue structure for large-scale discrete event simulation. *Transactions on Modeling and Computer Simulation*, 15(3):175–204, 2005.

[25] R. K. Treiber. Systems programming: coping with parallelism. Technical report, IBM US Research Centers, Yorktown, San Jose, Almaden, US, 1986.

[26] J. G. Vaucher and P. Duval. A comparison of simulation event list algorithms. *Communications of the ACM*, 18(4):223–230, 1975.

[27] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 211–220. IEEE Computer Society, 2012.