A Lock-Free $O(1)$ Event Pool and its Application to Share-Everything PDES Platforms

Romolo Marotta, Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia
DIAG—Sapienza Università di Roma
romolo.marotta@gmail.com, {mianni,pellegrini,quaglia}@dis.uniroma1.it

***Abstract*—The large diffusion of highly-parallel shared-memory multi-core machines has led Parallel Discrete Event Simulation (PDES) platforms to a shift towards a *share-everything model*. This model is based on loose coupling between simulation objects and threads, lasting (as an extreme) no more than the lifetime of individual events. Concurrent threads can therefore CPU-dispatch events destined to any object at any point in time, thus fully sharing the workload of events to be processed on a fine grain basis. This demands for efficient mechanisms to share the overall pool of pending events by enabling parallelism in insertion and extraction operations. In this article we present a lock-free event pool which also provides amortized $O(1)$ time complexity for both insertions and extractions. It can sustain highly concurrent accesses, while not leading to noticeable performance degradation when scaling up the thread count. Experimental results demonstrate that our solution stands as a core facility capable of further raising up the pragmatical impact of such an emerging share-everything PDES paradigm.**

I. Introduction

The historical design approach of Parallel Discrete Event Simulation (PDES) platforms accounted for different threads (or processes) in charge of taking care of the execution of disjoint subsets of simulation objects. However, the advent and increasing diffusion of multi-processor/multi-core machines has led to a shift towards a of *share-everything* paradigm, where a thread can in principle take care of executing any concurrent simulation object, at any point in time. Examples of PDES platforms adhering (or approaching) to this paradigm can be found in [1]–[4].

A relevant innovation by share-everything PDES is that event pools are no longer accessed by threads in isolation, rather in a (fully) shared mode. This is required to enable threads not currently busy with some object to concurrently extract from the shared event pool the higher-priority (i.e. lower-timestamp) pending events, bound to whichever simulation object. This allows delivering computing power to higher-priority pending simulation work along the whole lifetime of the model's execution. Still, threads can concurrently access the pool to post newly scheduled events resulting from processing activities at the involved objects. As a consequence, the event-pool data structure has become a performance-critical component along a new dimension, which is represented by the level of concurrency of no-longer isolated accesses.

In this article we present a lock-free concurrent event-pool data structure tailored for share-everything PDES platforms, which enables unleashed parallelism of enqueue/dequeue operations. Unlike existing non-blocking data structures suited for managing pools, such as non-blocking (skip)lists [5], [6], which pay a linear (or at least logarithmic) cost for insertion operations, our solution provides $O(1)$ amortized time complexity. Also, concurrent accesses' non-blocking synchronization is guaranteed in our proposal by relying only on conventional facilities offered by the underlying ISA (Instruction Set Architecture), such as the Compare-and-Swap (CAS) machine instruction. This makes our solution of wide applicability in a variety of off-the-shelf machines by different vendors.

We released our non-blocking $O(1)$ event pool as free software[1], and we have also integrated it with a last generation share-everything open source PDES system. Further, we present a performance study showing the effectiveness of our proposal in differentiated settings. We conducted our experiments on a 32-core HP ProLiant machine, equipped with 64 GB of RAM, which outline excellent scalability of our lock-free $O(1)$ event-pool data structure up to the maximum count of physical processing elements in the underlying machine.

The remainder of this article is structured as follows. In Section II we discuss related work. The lock-free $O(1)$ event pool is presented in Section III. Section IV provides experimental results.

II. Related Work

Several data structures for event pools have been proposed in the literature. The Calendar Queue [7] is a timestamp-ordered data structure based on multi lists, each one associated with a time bucket, offering amortized constant time insertion of events with generic timestamps and constant time extraction of the event with the minimum timestamp. The Ladder Queue [8] 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

[1]Source code available at https://github.com/HPDCS/NBCQ.

bucket exceeds a given threshold. The LOCT Queue [9] 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 to rely on a global lock for serializing the accesses, which would be detrimental to scalability, as shown in [10].

The work in [11] 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 [12].

As for non-blocking management of sets by concurrent threads, various proposals exist (e.g., non-blocking linked lists [5] or skip-lists [6]), which anyhow do not offer constant-time operations. The non-blocking 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 non-blocking 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 non-blocking 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 bucked 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). These problems are completely avoided with our proposal since we provide truly amortized $O(1)$ time complexity joint to non-blocking operations, including the reshuffle of the bucket width.

Non-blocking operations in combination with constant time complexity have been studied in [10], 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 [1], [2] 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 [14]. 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 [3] 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 of 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.

Overall, compared to literature results, our proposal is the unique that jointly offers: i) non-blocking concurrent accesses, ii) amortized $O(1)$ operations via non-blocking dynamic resize of the buckets' width, iii) total order while managing timestamped event records, and iv) independence from specific hardware support.

III. The Lock-Free O(1) Event Pool

Our lock-free event-pool data structure is inspired to the Calendar Queue [7]. It is a non-blocking priority queue whose schematization is shown in Figure 1. The logical (simulation) time axis is partitioned into a sequence of slots, called *virtual buckets*, which are then mapped to the entries of a circular array—the *calendar*. Each virtual bucket is therefore associated with a *logical time span* (the bucket width) that determines what events should be placed into a given virtual bucket.

Each bucket is the head element of a non-blocking linked list realized as in [5], where one bit in the pointer to the next node is used to indicate whether a node has been logically deleted—it is still linked to the list, but it must be considered as already extracted by some concurrent (or already-finalized) operation. We rather exploit two bits of the pointer to the next element to introduce 2 additional per-node states, to indicate whether nodes have to be moved/ validated, as we shall discuss in the reminder of the paper. This is to allow non-blocking dynamic reorganizations of the calendar bucket size, which we enable in our solution while still permitting regular operations to be concurrently processed.

To ensure consistency of concurrent accesses to our data structure, we rely on two different Read-Modify-Write (RMW) instructions, namely *Compare-and-Swap* and *Fetch-and-Add*. The former atomically updates a given memory location if its current value is equal to an input value provided to the instruction, otherwise the update fails. The

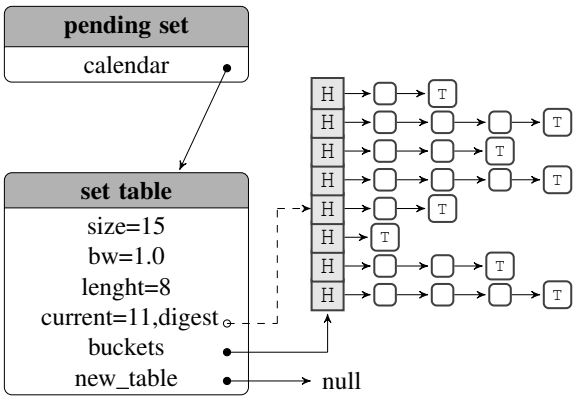

Figure 1. Conceptual organization of the non-blocking event pool.

latter allows to atomically retrieve the content of a memory location and increment its value. Given the possibility of failures for the Compare-and-Swap instruction, we rely on *retry cycles* to let an operation based on this machine instruction to be executed again, until it succeeds. This is typical of non-blocking algorithms.

As shown in Figure 1, our event pool relies on a pointer to a metadata table, referred to as *set table*, which is organized as follows. The `buckets` field points to the circular array which keeps the heads of the non-blocking linked lists. `size` tells how many elements are globally present within the set. `bw` determines the current bucket width, while `length` keeps the size of the circular array. `current` is the "index" of the virtual bucket from which the last element was extracted or a new minimum has been inserted—the actual bucket can be simply computed as `current` mod `length`. The `new_table` field is a pointer used to setup a new version of the calendar. We will refer to the creation of a new version as the *resize operation*.

Upon the execution of an enqueue or a dequeue operation, we first execute the READTABLE procedure, shown in Algorithm 3. The goal of this procedure is to obtain a reference to a *valid* set table, and to check whether some resize operation of the calendar is currently taking place. If a valid set table is found, enqueue and dequeue operations can continue. Otherwise, their execution is deferred until the current resize operation is completed. In the meanwhile, the deferred thread participates to the ongoing resize operation.

The pseudocode for the ENQUEUE() operation is shown in Algorithm 1. This operation takes an event e associated with timestamp T_e as a parameter. After having retrieved a valid set table using Algorithm 3, the operation determines what is the virtual bucket the event e being inserted belongs to, computed as $\lfloor \frac{T_e}{BW} \rfloor$, where BW is the (current) bucket width. Now, let B be the number of actual buckets of the calendar. Then, the event e is placed into the i-th bucket, where $i = \lfloor \frac{T_e}{BW} \rfloor \mod B$—according to the classical Calendar Queue organization, multiple virtual buckets are associated with the a same physical bucket entry.

Algorithm 1 Non-blocking ENQUEUE

```
1:  procedure ENQUEUE(event e)
2:      tmp ← new node(e)
3:      dig ← DIGEST()
4:      repeat
5:          h ← READTABLE()
6:          nc ← ⌊ n / h.bw ⌋
7:          bucket ← h.table[nc mod h.length]
8:          ⟨left, right⟩ ← bucket.SEARCH(tmp.t, VAL| MOV)
9:          tmp.next ← right
10:     until CAS(&left.next, UNMARK(right), tmp)
11:     repeat
12:         old ← h.current
13:     until nc > old.value ∨ CAS(&h.current, old, ⟨nc, dig⟩)
14:     Fetch&Add(&h.size, 1)
```

As already hinted, the i-th bucket is a non-blocking linked list, for which we assume the availability of the search procedure already defined in [5]. When executed, this procedure returns a couple of nodes, called *left* and *right* nodes. The SEARCH() operation tries to identify a coherent snapshot of the list, despite concurrent accesses, so that the left and the right nodes point to the nodes which would "surround" the new node. To cope with events with the same timestamp (not supported by the original proposal in [5]) we associate each event record with both a timestamp and a sequence number, which is always unique for all concurrent events. Indeed, an event has its sequence number set to zero, unless other concurrent events are present. In this latter case, the left node for the ENQUEUE() operation will point to the node associated with the highest sequence number so far in the batch of concurrent events. The newly-inserted node gets its sequence number augmented by one unit, making it represent concurrent events' insertion order—our proposal therefore provides a total order of the elements in the calendar. Monotonicity of sequence numbers is guaranteed under concurrency scenarios thanks to the insertions using Compare-and-Swap instructions. As a final note, the sequence number can be abstracted as being one component of the timestamp, thus leading our data structure to always manage records associated with different timestamp values. We will implicitly assume such an abstraction in the remaining part of the presentation.

Compare-and-Swap is also used to make concurrent executions of enqueue/dequeue operations safe, independently of event timestamps' and sequence numbers' management. Therefore, we rely on a Compare-and-Swap instruction to update the next pointer of the "future" previous node atomically. This is reflected in line 10 of Algorithm 1. Upon finalizing the ENQUEUE() operation, the `size` field of the set table is atomically increased by using a Fetch-and-Add. We exploit this field to resize the calendar, in case the fetched value of `size` fires the triggering condition, as we shall discuss.

The pseudocode of the DEQUEUE() operation is shown in Algorithm 2. Initially, a valid `current` should be taken

Algorithm 2 Non-blocking DEQUEUE

```
1: procedure DEQUEUE( )
2:     while true do
3:         h ← READTABLE()
4:         oldCur ← h.current
5:         cur ← oldCur.value
6:         bucket ← h.table[oldCur.value mod h.t_size]
7:         ⟨left, right⟩ ← bucket.SEARCH(0, VAL | MOV)
8:         rNext ← right.next
9:         newCur ← h.current
10:        if (newCur ≠ oldCur ∧ newCur.value ≤ cur) then
11:            continue
12:        else if ISMARKED(left.next,MOV) then
13:            continue
14:        else if ¬ISMARKED(rNext, VAL) then
15:            continue
16:        else if right = tail∧ h.t_size = 1 then
17:            return null
18:        else if right.ts < cur · h.bw then
19:            CAS(&h.current, cur, ⟨⌊right.ts/h.bw, DIGEST()⟩⌋)
20:        else if right.ts ≥ (cur + 1) · h.bw then
21:            CAS(&h.current, cur, ⟨cur + 1, DIGEST()⟩ )
22:        else if CAS(&right.next, rNext, MARK(rNext)) then
23:            Fetch&Add(&h.size, -1)
24:            return right.event
```

from the valid set table. Therefore, similarly to ENQUEUE(), the DEQUEUE() operation relies on READTABLE(). We again resort to the SEARCH() procedure defined in [5] to find the first node which is valid, yet we specify a "wildcard" timestamp set to zero as the priority for the search. In this way, we are sure that the left node will point to the head of the list, while the right node will point exactly to the node with minimum timestamp in the list.

Nevertheless, we must ensure that the right node belongs to the current *virtual* bucket, since the list associated with a physical bucket entry can span multiple virtual buckets. To this end, we check the timestamp of the node, and if it falls in the simulation-time span covered by the current virtual bucket, we can attempt to extract it. In the negative case, the right node belongs to a different year of the calendar, and we therefore switch to the correct bucket. This happens as well in case the right node is the tail of the list, since it only represents an overflow node required by the non-blocking list management.

Our event pool is designed to be independent of the nature (speculative vs conservative) of the share-everything PDES platform it would be integrated with. Hence, differently from the traditional Calendar Queue, we support insertions associated with simulation time intervals in the past of the `current` bucket. If `current` has been updated by a concurrent ENQUEUE() operation, then DEQUEUE() is retried in our design. This choice is dictated by correctness motivations (linearizability) and helps as well to deliver the highest priority (say lowest timestamp) event whose concurrent insertion in the event pool has been already materialized. This can be relevant for actual implementations of share-everything PDES platform, as discussed in [1]. Also, all the Compare-and-Swap instructions that target `current` should emulate Load-Link and Store-Conditional instructions so as not to loose updates by concurrent operations. This is done in our implementation by relying on a couple $\langle thread\_id, counter \rangle$ which is installed by a thread as a field of `current` (represented as a digest) each time a successful Compare-and-Swap instruction succeeds.

Similarly to [5], we handle DEQUEUE() operations by trying to mark the right node as invalid, by setting one bit in the next pointer. This logically downgrades the shared node to a thread-private status, meaning that the thread running the DEQUEUE() can safely manipulate its content. To ensure consistency, this is atomically done by using a Compare-and-Swap. While trying to extract the event with the highest priority, the Compare-and-Swap might fail due to concurrent operations, and the DEQUEUE() is simply retried. If none of the above conditions forces the DEQUEUE() operation to restart, the minimum timestamp element has been successfully identified and downgraded to thread-private status, thus it can be safely returned for usage by the overlying PDES engine.

The amortized $O(1)$ time complexity of insertion and deletion operations is obtained by ensuring that, on average, the number of elements within each bucket is balanced, similarly to the classical Calendar Queue. To this end, when we detect that the number of elements in the buckets is no longer balanced, we execute the *resize* operation. As mentioned, this is triggered by the value of `size` upon a queue operation, if the number of total events is over or below a certain threshold. To carry on a resize operation, we first "freeze" the current valid table. To this end, we allocate a new set table, and we publish a pointer to it into the `new_table` field of the old set table. Therefore, any thread operating on it will know that a resize operation is taking place, and will start to participate.

Before moving items from the old table to the new one, we mark each entry of the bucket array and the first nodes of the associated non-blocking lists as `MOV` (exploiting the aforementioned 2-bit status information within the pointer to the next node). This guarantees that dequeue operations are restarted any time that a node marked as `MOV` is encountered. This prevents dequeueing nodes while a resize operation is being executed, and allows the threads executing such dequeues to join the resize operation.

We then determine a new bucket width and a new length of the bucket array according to the strategy proposed in [7]. In particular, we scan a certain amount of events, depending on the total number of events placed in the queue, and we compute the *average timestamp separation*. This is the average distance, on the simulation time axis, between each couple of events, and it can be used to determine the new bucket width. The result is stored with a Compare-and-Swap in the `bw` field of the `new_table`.

Clearly, we want to achieve non-blocking properties also during the resize operation. Our strategy to reach this target

is based on flagging nodes to be migrated towards the new installation of the calendar as MOV nodes. Each thread that successfully flags as MOV a node to be migrated (via Compare-and-Swap) or finds a node as already marked, then allocates a new node instance to be linked to the new installation, as a copy of the original node. Then it enqueues it in the new installation by leaving it initially marked as invalid (INV) to prevent its extraction. After, the original node to be removed from the old calendar is flagged with the address of the new node instance in the new installation. At this point the node in the new installation is moved from the invalid state to the valid one (VAL), and the original copy is removed from the old installation. All these operations are still based on Compare-and-Swap. This migration logic is motivated by the fact that a thread performing a migration operation of a node might be delayed (e.g. because of a reschedule on CPU). This does not lead to blocking scenarios in our implementation since any other thread that is still running the migration can take care of trying to migrate a same node originally targeted by the delayed thread, and can take care of finally flagging the nodes in the old and in the new versions of the calendar. In fact, if a thread tries to move again a node flagged as MOV, which has already been inserted in the new installation of the calendar, the move will fail since the timestamp associated with the node is already found to be in the new calendar version, and the thread will simply take care of aligning the flags of the two buffer instances (old and new). Clearly, the number of nodes to be still migrated will eventually be equal to zero, a condition that will be detected by simply finding all buckets empty.

A. Garbage Collection

After that a node is marked as logically deleted by a thread, we do not know whether other threads are using the same buffer for, e.g., list traversal. In our approach, to safely reclaim event buffers, each thread maintains a couple of private lists of to-be-freed nodes, namely *old* and *new* lists, and an array of T flags, where T is the number of threads. Whenever a node is extracted from the queue or a `table` is swapped with a new one, the corresponding buffer is connected to the new list, and the thread updates its entry in the arrays. In this way, we never lose a reference to a memory buffer. If a thread reads every flag in its array as set, it releases every pointer in the old list and swaps the old and the new lists, also resetting the array of flags. This check is done periodically in our implementation.

IV. Experimental Results

We experimentally assessed our proposal in two different scenarios. The first is a stand-alone evaluation of the non-blocking $O(1)$ event pool, which has been based on a workload adhering the well known *Hold Model* [15]. In the second scenario we tested the non-blocking $O(1)$ event pool when integrated within an open source share-everything PDES environment. This allowed us to assess its benefits when employed within a real parallel simulation framework. All the tests have be run on 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.

Algorithm 3 Non-blocking READTABLE

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

A. Experiments with the Hold Model

The Hold Model is devised to emulate and evaluate the steady-state behavior of event pools. It is based on pre-populating the event pool with a given (parametric) number of events and on performing a sequence of dequeue/enqueue operations. In our tests, each concurrent thread performs either an enqueue or a dequeue with equal probability set to 0.5 (Markov Hold Model). Each run ends when the total number of performed operations (across all the concurrent threads) reaches 10^6. This guarantees the highest concurrency degree (depending on the selected number of threads) along the whole lifetime of the run, since no thread is ever switched off before the ending condition is reached.

We used four different priority increment distributions for

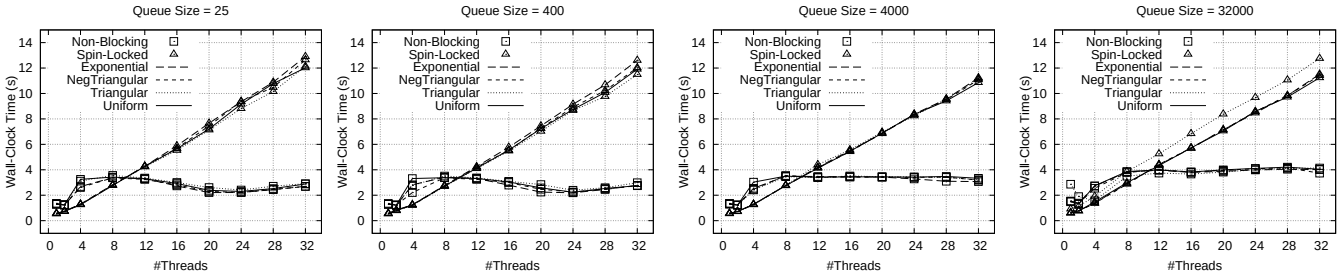

Figure 2. Results with the Hold Model.

the generation of the timestamps of new events to be inserted in the event pool, namely uniform, triangular, negative triangular and exponential. Also, we generated (for each considered distribution) 4 different tests, each one associated with different amounts of events pre-filled in the event pool, namely 25, 400, 4000, and 32,000. This allowed us to assess our proposal in settings resembling either small models (say with reduced number of activities to be performed, which are already posted into the event pool) and larger ones. The performance achieved by our non-blocking $O(1)$ event pool has been tested against the one achievable by relying on a classical Calendar Queue, with concurrent accesses synchronized by a spin-lock to make them consistent.

The results for all the tests we performed are shown in Figure 2, where each reported sample results as the average over 10 different runs of a same configuration. In particular, we plot the wall-clock time for carrying out the target number of event-pool operations while varying the number of employed threads between 1 and 32. From the results, we can draw two main conclusions. First, similarly to the Calendar Queue, our non-blocking $O(1)$ event pool is able to deliver performance that (once fixed the number of threads) is essentially independent of both the event timestamps' distribution and the number of pre-filled events (the queue size). This is an indication of excellent capability of dynamic reorganization of the bucket width in our solution. Also, the wall-clock time curves are essentially flat while scaling up the number of concurrent threads, which indicates how our proposal is resilient to performance degradation phenomena caused by conflicting accesses when increasing the level of concurrency in the event-pool operations. As somehow expected, flatness of the wall-clock time is not achieved by the spin-lock protected Calendar Queue, which leads to performance degradation that is linear versus the number of employed threads. Additionally, the absolute performance of the spin-lock protected Calendar Queue becomes worse than the one of our non-blocking $O(1)$ event pool as soon as the number of concurrent threads oversteps the value 8/12.

B. Experiments with a share-everything PDES platform

We have integrated our proposal with the share-everything PDES engine standing at the core of the RAMSES speculative simulation framework [1]. In this engine, a meta-data layer is used to keep track of what simulation object is currently being run by any thread, and of the corresponding event timestamp. Meta-data are updated by threads upon extracting events from an underlying event pool, which is fully shared by all the threads. These meta-data are used to compute a reduction to assess what event is associated with the current commit horizon of the simulation, which can be therefore safely processed, with no need for reversibility of the state updates it performs. This is achieved by the worker thread by triggering a native version of the application code, not including support for squashing the performed computation. If the event is not safe (i.e. events in its past could still affect it), then the worker thread runs a modified version of the application code, that is transparently instrumented (by an ad-hoc compile/link procedure) in order to generate at run-time so called undo code blocks, which are minimal blocks of machine instructions that can be used to revert the updates performed by the event processing phase. The simulation model lookahead is exploited at the engine level in order to determine what other events—beyond the one associated with the commit horizon of the simulation—can be processed safely, since they will not eventually be affected by causality errors. In this engine, the fully-shared event pool only keeps so called *schedule-committed* events, namely those that are generated by events that will never be rolled back. Hence if a thread processes an event speculatively, then the produced output events are kept buffered outside the event pool up to the point in time where the event becomes safe (thanks to the advancement of the commit horizon). They are simply discarded—with no inclusion at all in the shared event pool—if the original event is eventually rolled back. Conflicting accesses by multiple threads to the same simulation object, say because of extraction from the event pool of two or more events destined to the same object, are resolved via a read/write spin-lock approach giving higher priority to lower timestamp events. In the original implementation of this engine, the shared event pool residing at the lowermost layer was a Calendar Queue protected via a spin-lock.

As test-bed application we used PHOLD [16], configured with 1024 simulation objects. Each object schedules two

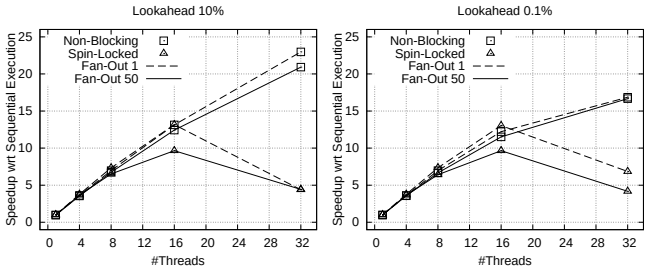

Figure 3. Results with PHOLD.

different types of events, *regular* and *diffusion* events. Both types of events resemble classical PHOLD events, as their processing leads to spending some CPU time, via a busy loop emulating a given event granularity. The difference among regular and diffusion events lies in that the former class generates events, while the latter does not generate any other event. Both types of events lead anyhow to perform updates on the state of the target simulation object. In particular, the updates involve statistics related to the advancement of the simulation, such as the number of events processes (and of which type) and average values (as well as peaks) of the simulation time advancement experienced by the objects when processing new events. Such an event pattern leads to the scenario where the average number of events (of any type) hosted by the shared event pool does not change along time, but we experience punctual fluctuations. The selection of this event pattern is an explicit choice aimed at observing how burst insertions of diffusion events in the event pool, which are generated when processing regular events, lead to run-time dynamics (possibly) affected by phase-intensive accesses to the shared event-pool data structure. In our tests we set the number of diffusion events generated by a regular event, referred to as Fan-Out, to the values 1 and 50. The latter settings leads to scenarios with higher intensity of the burst of insertion operations of new events in the shared pool upon committing some regular event. Also, it well mimics PDES dynamics proper of epidemic models. As for the timestamp increments when generating new events, we selected an exponential distribution with mean set to one simulation-time unit. As for the CPU requirements by the events, we set it on the order of 60 microseconds, which emulates low to medium granularity settings, which are proper of a vast variety of discrete event models. Further, we selected two different lookahead values, namely 10% and 0.1% of the average timestamp increment, so as to observe run-time dynamics under different patterns in relation to safe vs speculative processing.

In Figure 3 we report the speedup achieved when running the shared-everything PDES engine with different number of threads up to 32, compared to the case of execution with a single thread. We still took the spin-lock protected Calendar Queue as the baseline for performance comparison in this study. By the data we observe how our non-blocking $O(1)$ event pool allows for close to linear speedup for lookahead value set to 10% of the average event timestamp increment, with coefficient 1 (ideal speedup) when running up to 8 threads, and with coefficient at least 0.7 when running with higher concurrency (i.e. up to 32 threads). The curve referring to Fan-Out 50 stands quite close to the one with Fan-Out 1, in fact the two speedup curves differ by slightly more than 10% only when running with 32 threads. This is a clear indication that our non-blocking event pool is able to efficiently handle scenarios with bursts of (concurrent) operations—just depending on the pattern according to which (sets of) new events are produced by the application code when processing some event. This is not guaranteed by the spin-lock protected Calendar Queue that, beyond showing scalability problems when running with more than 16 threads, also shows a clear decrease of performance for the scenario with Fan-Out set to 50 and thread count larger than 8. In fact, spin-lock based accesses to the event pool are adverse in scenarios where more intense bursts of enqueue operations occur, a phenomenon that with Fan-Out set to 50 becomes evident as soon as the concurrency level in the access to the pool is non-minimal. Very similar considerations can be drawn for the experiments with definitely reduced lookahead, set to 0.1% of the average event timestamp increment. Also, with such a reduced lookahead value, more/longer synchronization phases across threads take place at the level of the synchronization meta-data management layer within the PDES engine (since lower lookahead leads to reduced likelihood of processing safe events, and to higher likelihood of delayed commit for speculatively executed events). As a consequence, a larger percentage of time is spent by the threads within the synchronization layer managing causality meta-data, which leads to a slightly reduced pressure in the access to the shared event pool. This allows the spin-lock protected Calendar Queue to achieve a bit higher speedup (compared to lookahead set to 10%) when the Fan-Out parameter is set to 1. For our non-blocking $O(1)$ event pool, the reduced pressure leads to achieve the same speedup with the two different Fan-Out values (1 or 50) even when running with 32 threads (at this point the speedup is still 0.55 of the ideal one). This indicates higher potential for performance benefits by our solution, compared to the spin-lock protected Calendar Queue, in scenarios where the pressure of event pool accesses (slightly) diminishes.

V. Conclusion

In this article we have presented a lock-free event pool offering $O(1)$ amortized time complexity of concurrent insertion/extraction operations. Our proposal is well suited for emerging organizations of PDES platforms to be hosted on top of shared memory multi-core machines, where concurrent threads are allowed to (fully) share the workload of

events to be processes on a very fine grain basis. Hence a fully-shared event pool, guaranteeing high throughput of concurrent operations joint with the delivery of higher priority events upon extractions stands as a core building block for the improvement of the performance delivered by this kind of *share-everything* PDES platforms. We have also reported experimental data demonstrating the efficiency of our proposal.

ACKNOWLEDGEMENTS

Mauro Ianni and Alessandro Pellegrini are also working with Value Up S.r.l., an InResLab partner. This work is partially supported by the REWIND: "Techniques and support for software reversibility" project funded with the support of MISE research funds.

REFERENCES

[1] 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*, ser. PADABS, S. Hunold, A. Costan, D. Ginenéz, A. Iosup, L. Ricci, M. E. Gómez Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer, and M. Alexander, Eds. LNCS, Springer-Verlag, 2015, pp. 466–478.

[2] E. Santini, M. Ianni, A. Pellegrini, and F. Quaglia, "Hardware-transactional-memory based speculative parallel discrete event simulation of very fine grain models," in *Proceedings of the 22nd International Conference on High Performance Computing*, ser. HiPC. IEEE, dec 2015, pp. 145–154.

[3] 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*, ser. PADS. ACM Press, 2015, pp. 75–86.

[4] 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*. ACM Press, 2013, pp. 103–114.

[5] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC, J. Welch, Ed. Springer Berlin/Heidelberg, 2001, vol. 2180, pp. 300–314.

[6] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609–627, may 2005. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0743731504002333

[7] R. Brown, "Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem," *Communications of the ACM*, vol. 31, no. 10, pp. 1220–1227, 1988.

[8] 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*, vol. 15, no. 3, pp. 175–204, 2005.

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

[10] 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*, ser. PADS. ACM Press, 2014, pp. 15–26.

[11] R. Ayani, "LR-Algorithm: concurrent operations on priority queues," in *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, ser. SPDP. Dallas, TX, USA: IEEE Computer Society, 1990, pp. 22–25.

[12] R. Rönngren and R. Ayani, "A comparative study of parallel and sequential priority queue algorithms." *Transactions on Modeling and Computer Simulation*, vol. 7, no. 2, pp. 157–209, 1997.

[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 ICST Conference of Simulation Tools and Techniques*, ser. SIMUTools. ICST, 2016.

[14] D. R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and System*, vol. 7, no. 3, pp. 404–425, 1985.

[15] J. G. Vaucher and P. Duval, "A comparison of simulation event list algorithms," *Communications of the ACM*, vol. 18, no. 4, pp. 223–230, apr 1975.

[16] R. M. Fujimoto, "Performance of Time Warp Under Synthetic Workloads," in *Proceedings of the Multiconference on Distributed Simulation*. Society for Computer Simulation, 1990, pp. 23–28.