Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

A low-overhead constant-time Lowest-Timestamp-First CPU scheduler for high-performance optimistic simulation platforms

Francesco Quaglia

DIAG – Sapienza Universita' di Roma, Italy

A R T I C L E I N F O

Article history:
Received 10 September 2014
Received in revised form 9 January 2015
Accepted 13 January 2015
Available online 26 February 2015

Keywords:
Parallel discrete event simulation
Optimistic synchronization
CPU-scheduling
Performance optimization

A B S T R A C T

An approach to high-performance discrete event simulation consists of exploiting parallelization techniques. These rely on partitioning the simulation model into multiple, interacting simulation objects, also known as Logical Processes (LPs), which concurrently execute events on different CPUs and/or multiple CPU-cores. However, despite the tendency towards high degree of hardware parallelism, for relatively large models, multi-programming schemes are still needed in order to share a single CPU-core across multiple LPs. Consequently, priority management and CPU-scheduling remain central issues for the effectiveness of any parallel simulation environment. This article focuses on the optimistic approach to parallelism, which is based on speculative processing and maintains event-causality across concurrent LPs via rollback techniques. Specifically, the article presents a low-overhead constant-time implementation of the well known Lowest-Timestamp-First algorithm for the identification of the next LP to be CPU-dispatched. This proposal is suited for contexts where the optimistic simulation system conforms to the best-practice of keeping separate event lists for the hosted LPs. The implementation has been integrated in the open source ROOT-Sim (ROme OpTimistic Simulator) package. The effectiveness of the presented proposal is assessed via an extended performance study, carried out by relying on the *game of life* as the test-bed application.

1. Introduction

Discrete Event Simulation (DES) is a well known technique to study and predict the behavior of complex systems. This technique has also acquired the role of core component in scenarios entailing critical decision making processes, such as for symbiotic systems and for on-line reconfiguration of systems based on (faster-than-real-time) what-if analysis. This has renewed the interest in solutions allowing quick delivery of simulation results to end-users or applications.

A traditional way to achieve high performance discrete event simulations is the employment of parallelization techniques, globally referred to as Parallel-DES (PDES) [1]. They are based on the partitioning of the simulation model into Logical Processes (LPs) that can execute events in parallel on multiple CPUs and/or CPU-cores, and rely on synchronization mechanisms to achieve causally consistent execution of simulation events at every LP. More in details, the classical consistency criterion is the one according to which the execution trajectory of state updates at each LP needs to reflect timestamp-ordered processing of all the simulation events that have been destined to the LP (including the events produced as the result of processing activities by any other LP).

E-mail address: quaglia@dis.uniroma1.it

http://dx.doi.org/10.1016/j.simpat.2015.01.009
1569-190X/© 2015 Elsevier B.V. All rights reserved.

The optimistic synchronization approach [15] allows each LP to process its events without previous assessment of their causal consistency. Hence an LP that processes and event at timestamp t may later receive an event with timestamp $t' < t$. To recover from this non-consistent execution, rollback schemes are employed. This synchronization approach is likely to favor speedup in general application/architectural contexts. In particular, it has been shown to exhibit performance relatively independent of the lookahead of the specific simulation model, and has also been shown not to suffer (in terms of amount of rollback in the parallel execution) from non-minimal message delivery latency. The latter feature makes it suited for a wide variety of computing platforms, including large scale traditional GRID systems and desktop GRID environments [2].

One core issue to address in the design of efficient optimistic simulation platforms deals with the selection of the next LP to be dispatched in scenarios where multiple LPs share the same CPU-core (i.e. they are executed along the same thread). This is the typical case for (very) large models, where the number of LPs can be (significantly) greater than the number of available CPU-cores within the computing platform. As for this issue, several CPU scheduling policies have been proposed in literature, which are aimed at optimizing performance of optimistic simulation systems. Among them, we can mention policies based on a mixture of timestamps and event granularity information [3], or on statistical inference on the timestamp distribution of the events along the simulation time axis [4].

However, the most commonly adopted policy is Lowest-Timestamp-First (LTF) [5], according to which the next LP to be dispatched is the one whose next event has the minimum timestamp across all the LPs sharing the same CPU-core. LTF is attractive for its simplicity, and especially because it does never cause out-of-timestamp-order event processing across those same LPs. To better clarify this aspect, we depict in Fig. 1 an example scenario where LP_a and LP_b are supposed to execute their events along the same thread (hence running on the same CPU-core). When processing its event e_x with timestamp 3, LP_a produces a new event e_y with timestamp 5 destined to LP_b. On the other hand, LP_b has an event e_z to be processed with timestamp 10. If LP_b were dispatched prior to LP_a for processing e_z along the thread in charge of handling execution of both LP_a and LP_b, we would have incurred a timestamp-order violation upon the successive processing of e_x by LP_a, given that the final effect is the production of an event destined to LP_b, namely e_y, with timestamp in the past of e_z. This scenario is avoided by LTF given that this algorithm leads to dispatch LP_a for processing e_x prior to dispatching LP_b for processing any of its events. Hence event e_y would then be available for processing by LP_b on time, and this LP would then be allowed to process its events e_y and e_z in correct timestamp order. Overall, with LTF causality errors and rollbacks can only be triggered by events exchanged between LPs hosted on top of different CPU-cores (i.e. run by different threads), which might ultimately yield to violate the non-decreasing timestamp-order rule for processing activities at some LP.

On the other hand, advanced optimistic simulation systems typically maintain different event lists, one for each LP (see, e.g., [6–8]). This is because the identification of past and future events, and the move of events from, e.g., past to future upon rollback, needs to be actuated on a per-LP basis for efficiency reasons. In such a scenario, the algorithm for identifying the future event with minimum timestamp according to LTF, requires adequate design/implementation in order to prevent the CPU scheduling task to become a bottleneck. A classical solution according to which the timestamps of the next events of different LPs are kept within an array requires complete array scanning upon the scheduling operation, with linear cost vs the number of LPs sharing a same CPU-core. On the other hand, organizing those timestamps within an ordered data structure, such as a heap, requires logarithmic cost for both scheduling (i.e. minimum element extraction) and/or heap-update upon variations of the next event timestamp for an LP.

In this article we present the design and implementation of an LTF scheduler that provides *constant-time average performance*, with very limited actual overhead. In other words, our solution allows constant time operations in the highly likely case. In fact, CPU-scheduling operations requiring non-constant-time may only occur upon a significative variation of the locality of the next-event timestamps along the simulation time axis. The statistical significance of such a situation clearly depends on the event pattern along simulation time for the specific application. However, there is an empirical evidence (see, e.g., [9,10]) that event patterns are, at any time, characterized by greater density of events in the near future of the actual *Global Virtual Time* of the optimistic run, namely the commitment horizon for already processed events, and by significantly

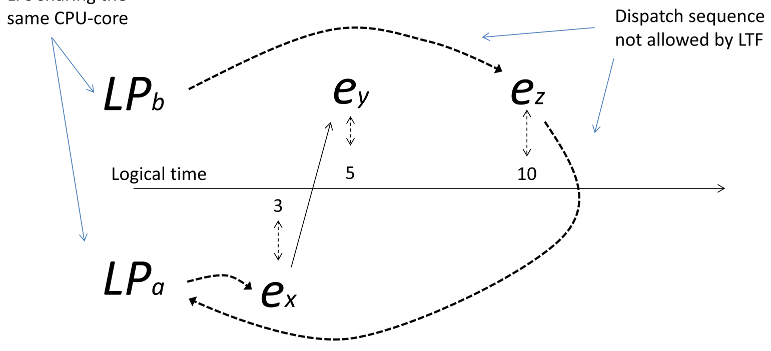
LPs sharing the same CPU-core
Dispatch sequence not allowed by LTF
LPb
ey
ez
Logical time
5
10
3
LPa
ex

Fig. 1. Example scenario.

reduced event density in the far future. As a consequence, the locality of LPs' next events along simulation time likely changes slowly. Overall, non-constant time scheduling operations represent the unlikely case for our LTF scheduler.

Our proposal can be seen as a variation of the priority queue for event sets known as Calendar-Queue [11]. However, the calendar queue offers constant-time average performance with actual limited overhead for element insertions/extractions in cases where the amount of elements is similar to, and is evenly distributed on, the amount of time buckets within the calendar. In cases where this assumption does not hold, constant-time is provided with non-minimal actual overhead. With our variation, we keep the constant-time average performance property by jointly ensuring minimal overhead without requiring a balanced distribution of the elements (namely the next-event timestamps of the LPs) over the time buckets, thus increasing the efficiency of the scheduler vs the actual distribution of the timestamps of the LPs' next events along the simulation time axis.

As for additional insights on the relation between our proposal and the Calendar-Queue algorithm, we note that Calendar-Queues have been widely exploited (and analyzed) in literature (see, e.g., [24]) in the contexts of sequential discrete event simulation systems. Particularly, they have been used to keep the whole (dynamic) set of pending-events to be sequentially (and in timestamp-order) processed by the simulation software. Calendar-Queues are also typically exploited to achieve optimized sequential runs (i.e. with low cost management of the pending-event set), to be used as the reference for evaluating the speedup achievable via parallel ones (see, e.g., [29]). In these scenarios, any event injected in the system (and hence recorded in the Calendar-Queue) does never need to be revoked, given that the timestamp-ordered sequential execution is always guaranteed to be causally consistent. In this article we leverage on the basic data structures used in the Calendar-Queue and propose a variant that is able to cope with scenarios where some registered element (representing the CPU-scheduling priority associated with the corresponding LP) can be cancelled, since it reveals as no more causally consistent, which is reflected into a dynamic change of the priority of the associated LP (and possibly into a dynamic change of the LP with the highest priority). This scenario is not one the original Calendar-Queue is tailored for.

As for the usage of Calendar-Queues in PDES systems, a recent trend lies in using a particular variant, called Ladder-Queue [25], as the support for CPU-scheduling operations of the LPs. In particular, the Ladder-Queue has been shown to represent a well suited data structure for managing the relative priority of the LPs in optimistic PDES systems based on the multi-threading paradigm [26]. We will explicitly compare our solution with the Ladder-Queue in our experiments, showing how our approach can further improve the performance delivered by the optimistic PDES system. Additional details on the relation between our proposal and both Calendar-Queues and Ladder-Queues are anyhow provided in the Related Work section of this article.

We present our LTF scheduler by initially outlining its design principles, which remain valid independently of the programming language and/or target hardware architecture. After, we provide hints on a specific implementation of the scheduler entailing optimizations suited for x86 architectures. Also, we release the implementation by integrating it within the open source ROOT-Sim (ROme OpTimistic Simulator) package [12], namely a run-time environment transparently supporting parallelization and optimistic processing for DES models relying on event handlers conforming to ANSI-C programming rules.

We also report the results of an extended experimental study demonstrating the effectiveness of our proposal when tested on the well known *game of life* model, with variation of the model size up to 10^5 LPs. The data have been gathered by running the experiments on top of a 32-core HP ProLiant machine equipped with 64 GB of RAM, which is representative of current off-the-shelf commodity hardware exploitable for scientific computing (such as discrete event simulation).

The remainder of this article is structured as follows. In Section 2, we recalls basic notions and terminology related to optimistic simulation systems. Related work is discussed in Section 3. In Section 4, we present the design/implementation of the LTF scheduler. Experimental data are provided in Section 5.

2. Optimistic simulation basics

In PDES each LP has its own view of simulation time advancement, tracked via the so called *Local Virtual Time* (LVT), and the LPs interact by exchanging messages, which are used as the means for notifying the scheduling of a simulation event (at a given time) for the recipient LP. Each message carries the content of the scheduled event (e.g. the type of the event) and the event timestamp, which represents the simulation time for the occurrence of that event at the destination. Any event execution updates the state of the corresponding LP by also moving the LVT of the LP to the event timestamp. It possibly schedules new events destined to whichever LP included in the simulation model partitioning. Timestamp ordered execution of simulation events at each LP is a sufficient condition for the correctness of simulation results [1,13] (although not a necessary one [14]), which is enforced via proper synchronization schemes.

In the optimistic approach to synchronization [15], events are stored into per-LP event-lists, each of which is logically partitioned into a *future-event-list* and a *past-event-list*. The future-event-list stores events not yet processed, while the past-event-list records already processed events. Each LP is eligible for CPU-scheduling and event-processing unless its future-event-list is empty. Once dispatched, the LP is allowed to process the event from the future-event-list having the minimum timestamp. Such an event is moved to the past-event-list once the LP is dispatched for processing it.

Timestamp order violations might arise since an LP may receive a message carrying an event with timestamp lower than its LVT (given that LP dispatching is not subject to – system wide – safety verification of causal consistency of its next to-be-processed event). If a timestamp order violation is detected, all the events that were executed out of timestamp order are rolled back (they are moved from the past-event-list to the future-event-list). Also, the LVT of the LP is pushed back to the timestamp of the last event executed in correct order, and the LP state is recovered to its value prior to the timestamp order violation, which is achieved either by relying on traditional checkpointing methods (see, e.g, [16–19]) or by the means of reverse computing approaches (see, e.g., [20]). As for the effects involving other LPs, which resulted from the execution of the events that are rolled back, these are undone by sending an *antimessage* for each scheduled event sent out during the rolled back portion of the simulation ([1]). Upon the receipt of an antimessage associated with an already executed event, the recipient LP rolls back as well. If the event has not yet been executed, the antimessage has the only effect to "annihilate" the event. After rolling back, the LP resumes execution of events from its future-event-list.

A relevant concept to optimistic synchronization is Global Virtual Time (GVT), which is defined as the smallest timestamp among those of (i) unexecuted events already inserted into the LP event lists, (ii) events being executed, (iii) messages/antimessages in transit. Since no LP can ever rollback to simulation time preceding GVT [15], the GVT value indicates the commitment horizon of the simulation. It is used both to execute actions that cannot be subject to rollback, such as displaying of intermediate simulation results, and for recovering memory. Specifically, events with timestamp lower than GVT will never need to be re-executed after a rollback, therefore they can be discarded by the past-event-lists of the LPs. The same happens to obsolete state information, if any, maintained to support state recoverability. The action of recovering memory after GVT calculation is typically referred to as *fossil collection*.

The reader can refer to literature works for additional details on PDES aspects and related synchronization methods, such as the recent survey provided in [33].

3. Related work

The processor scheduling problem in optimistic simulation systems has been deeply investigated in literature. In particular, several works have proposed solutions aimed at bounding the negative performance effects of rollbacks within the parallel/distributed execution of the simulation model. The solutions in [4,21–23] target the reduction of the amount of rollback by assigning priorities to the LPs according to some statistical criterion aimed at reducing the likelihood of timestamp order violations within the system. The work in [3] extends such a perspective by providing a scheduling framework targeting both the reduction of the amount of rollback, and the reduction of the waste of time associated with rolled back computations. However, due to its simplicity, the standard solution for the scheduling problem is still represented by LTF [5]. As hinted, this algorithm has the advantage of avoiding causality violations across the LPs sharing the same CPU-core (i.e. executed along the same thread), which contributes to keep low the amount of rollback while executing the simulation model. Compared to the above works, in this article we do not target the definition of any new algorithm. Instead, we provide a highly efficient implementation of the LTF standard algorithm.

Our approach has also relations with the literature on priority queues, and in particular with works addressing the maintenance of timestamp ordered event sets (see [24] for a comparative analysis). Among these works, the closest one to our approach is [11], where a so called Calendar-Queue algorithm is presented, which provides, as its average performance, constant-time insertions of events with generic timestamps, and constant-time extraction of the event with the minimum timestamp (i.e. the highest priority event). This solution is based on a linear data structure (an array) whose entries are associated with different time buckets, and the constant-time property is guaranteed with low overhead only in cases where the distribution of the event timestamps (in combination with the dynamic resize of the linear data structure) allows having a number of events similar to the number of time buckets, with even distribution of the events on the different buckets. Instead, in our proposal we use a combination of a linear data structure and a hierarchical one, which allows us to provide constant-time average performance with low actual overhead for all the cases in which the distribution of the timestamps of the next events of the LPs to be CPU-scheduled (in combination with the resize of the data structures) is such that each time bucket is associated with at most K elements (with K being a configurable parameter), and at least one element is registered (at any time) within a significatively large bunch of buckets. This is a statistical property less strict than the one required for actual efficiency of the Calendar-Queue [11], since it expresses the possibility of skews for the distribution of the timestamps of the different elements. Also, in the Calendar-Queue, insertion of elements with timestamp above the current upper-limit time bucket registered within the linear data structure is supported via a list with linear access cost. Instead, such an overflow scenario is handled in our scheduler via an overflow-table with $O(1)$ access cost. This can be achieved since our scheduler tackles a problem intrinsically different from the one tackled by traditional priority queues for event sets. These priority queues target scenarios where the number of elements registered within the priority queue can significantly change over time, while our scheduler is targeted to cases in which the maximum number of registered elements is fixed, or stable in time, since it corresponds to the number of LPs hosted by the underlying instance of the simulation platform ([2]). In fact, our target is to keep a single element for each LP (associated with the next to-be-processed event for that LP), which is used to discriminate the relative priority of that LP compared to the other LPs.

A variation of the Calendar-Queue exists, which aims at managing scenarios where skews in the distribution of the timestamps of the elements to be kept within the queue may arise, hence exhibiting capabilities related to the ones by our proposal. This is the Ladder-Queue presented in [25], where a second level bucket-array is exploited anytime the number of elements within a first-level bucket exceeds a predefined threshold ([3]). However, the target of the Ladder-Queue is to avoid

[1] An antimessage is an exact copy of the corresponding message, except for a single bit value.

[2] Variations may occur over time in case of, e.g., dynamic load balancing schemes migrating the LPs from one simulation kernel instance to another.

[3] Although up to 8 different levels of buckets are considered in the original presentation, in most of the actual implementations (see, e.g., [26]) a classical two level configuration is adopted.

excessively frequent resize of the first-level bucket array (and excessive usage of memory) in case of non-uniform timestamp distribution. Instead, our target is the reduction of the actual cost for both insertions and extractions of the elements, when also considering that an extraction may involve a generic element (not only the element with the minimum timestamp) just due to dynamic changes in the priority of the LPs. Overall, the Ladder-Queue copes with the typical sequential event schedule approach where, once scheduled, an event gets registered within the queue and is eventually extracted only upon becoming the one with the absolute minimum timestamp. Instead, in our scheduler, one element might need to be extracted before becoming the one with the minimum timestamp, exactly because the associated LP changes its CPU-scheduling priority over time depending on the arrival of newly scheduled events (or even antimessages) destined to it. Constant-time management of all these types of operations, with very reduced actual overhead, is achieved via the aforementioned hierarchical data structure, plus the proper overflow table (also exploited as a lookup table), which is fully complementary to second-level bucket arrays used by the Ladder-Queue. Experimental data for a comparative analysis, evidencing the performance benefits from our proposal compared to the Ladder-Queue, when used as the CPU-scheduling support in optimistic PDES systems, will be provided in Section 5.

Finally, our proposal has relations with solutions addressing process and thread scheduling in the context of Operating Systems (see, e.g., [27]). Among them, we can mention the Linux 2.6.x (or later versions) scheduler, which provides $O(1)$ CPU-scheduling capabilities. Compared to this scheduler, our proposal addresses the different situation where, endemically, the priority of an LP can change multiple times before the actual schedule of that LP occurs (e.g. because of dynamic variations of the next-event timestamp for that LP while the simulation model execution proceeds). This is not the case for the Linux scheduler, since any thread keeps its priority unchanged once registered as ready-to-run within the scheduler, and priority variations can only occur prior to re-entering the ready-to-run state (except for forced changes via, e.g., the `setpriority` POSIX system call). Another relevant difference is that the Linux 2.6.x scheduler does not require to discriminate across an a priori undetermined set of priority levels, thus admitting the possibility to register multiple threads at the same priority level as a common case. Instead, our LTF scheduler needs to keep track of (relative) priority values that are not prespecified to fall into given ranges and to manage a total order among the LPs' priorities (except for unlikely scenarios of identical timestamps for the next to-be-processed events of different LPs). This is reflected in that our implementation of LTF relies on a non-fixed size data structure, that dynamically adapts its capability to discriminate different priority levels depending on the event-timestamp pattern. Instead, the Linux scheduler relies on a fixed-size data structure keeping track of a predetermined, unchangeable amount of different priority levels.

4. The LTF scheduler

In this section we present the LTF scheduler design. We initially provide an overview of the data structures used to maintain the priorities of the LPs. Then we introduce the logic associated with the different operations taking place on these data structures. The latter entail both registering/deregistering an LP within the scheduler and also selecting the highest-priority LP currently registered within the scheduler. Time complexity of these operations is also discussed. After, we provide indications on how to enhance the above data structures (and the associated management logic) in order to cope with event patterns expressing significant variation of the locality of the next events of the LPs along simulation time and/or exhibiting significant skews in the corresponding distribution of their timestamps. This aspect includes the dynamical resize of the data structures in order to provide adequate space–time tradeoffs. Finally, details on our implementation based on the C language and tailored for x86 architectures are provided.

4.1. Data structures

Our LTF scheduler works by maintaining a tree-like data structure. Each leaf keeps information related to a fixed size simulation time bucket δ_j. This information consists of a double-linked list of entries, each one keeping the identity of an LP and the value of the timestamp of its next to-be-processed event (if any), in case it falls within δ_j. At any time, a single entry is registered within the data structure for any LP. On the other hand, an LP may not be registered within the scheduler at a given point in time during the execution of the simulation. This occurs in case no next to-be-processed event is currently present into the future-event-list of the LP, which means that the LP is not currently ready for CPU-dispatching.

The scheduler data structure is made up by several blocks, each one containing a representation of a tree with a same fixed arity and depth. Therefore, the *i*-th block keeps information about next-event timestamps falling within a fixed simulation time window Δ_i of size $|\Delta_i| = N \times |\delta_j|$, N being the number of leaves within the block. The tree associated with each block holds the information about the occupancy of its leaves by means of nodes forming a hierarchical bitmap. In particular, each node within the tree holds a summary of the status of each of its children (those being either other bitmap nodes, or leaves). This is done by logically linking the *i*-th bit of the bitmap of each bitmap node to the status of the *i*-th child, thus setting the bit to 1 if the child contains any element in its direct children or in its offspring, or to 0 if the branch contains no element in the associated leaves. Fig. 2 shows a schematization of such a data structure for the case of:

(i) two blocks, each one having a single occupied (and hence non-null) leaf;
(ii) trees organized as two-level bitmaps of size 2;
(iii) $|\Delta_i|$ set to 12 and $|\delta_j|$ set to 3.

In the example, we have three elements registered within the scheduler data structure, namely $\langle LP_i, 7\rangle$, $\langle LP_j, 16\rangle$ and $\langle LP_z, 17\rangle$, which indicates that three different LPs are ready for CPU-dispatching (hence, each of them can take control for processing its next simulation event). On the other hand, LP_i is the highest priority LP, given that its associated element has the absolute minimum timestamp among all the elements currently registered within the scheduler.

The index of the branch that contains the element with minimum timestamp within the simulation time interval covered by any block can be quickly retrieved via a simple operation of iteratively finding the first (most significant) set bit along the hierarchical organization of the bitmap nodes, starting from the root-node of the tree associated with that block. Also, by checking for a bitmap node to entirely be 0, which can be done efficiently in differentiated software technologies thanks to the reliance on, e.g., efficient logical instructions offered by the instruction-set of most of the conventional hardware architectures, we can quickly skip large parts of empty (sub) trees during the search operation.

Each tree (hence each block) is pre-allocated, in terms of bitmaps and leaves, within structured records. Consequently, once known the virtual address (or reference) where the structured record is located, the virtual address of any bitmap or leaf forming the tree can be determined by applying a displacement starting from that virtual address. Also, a top level indexing array `tli[]` is kept with an entry for each block, which provides a reference to the record associated with the tree related to that block (see again Fig. 2 for an example). Additionally, the index of the block currently containing the element with the minimum timestamp across all the LPs is explicitly kept in an apposite variable, which we refer to as `min_index`. For the example in Fig. 2, the current value of `min_index` is 0 since Block-0 is the one currently keeping the element $\langle LP_i, 7\rangle$, which is associated with the highest priority LP.

The data structure depicted so far can be seen as the expression of a one-way relation between a given simulation time interval δ_j and a set of LPs whose immediate future activities fall within this interval. To provide the scheduler with the ability to efficiently express the reverse relation, which starting from the identity of an LP determines the corresponding interval for its immediate future activities, the aforementioned data structure is complemented via a so called `LP_info[]` array, used as a lookup table. The *i*-th entry of this array keeps information about the timestamp of the element currently registered within the scheduler for the *i*-th LP, if any (see again Fig. 2). In this way, the interval δ_j for the immediate future activities of LP_i is determined by simply using that timestamp value as the hash key to determine the virtual address of the leaf associated with δ_j. This is done by dividing the timestamp value by Δ (used to denote the size of any generic time window Δ_k), so to retrieve the entry of `tli[]` associated with the block where the next to be executed event of that LP is registered, and then by dividing the remainder of the above division by δ (used to denote the size of a generic time bucket δ_j), so to determine the displacement to be applied to the identified `tli[]` reference in order to locate the leaf of interest.

Any entry `LP_info[i]` is also augmented with a reference field, namely `LP_info[i].elem`, which keeps the memory reference to the buffer hosting the element within the scheduler associated with the *i*-th LP.

4.2. Registering elements within the scheduler

Algorithm 1 reports the pseudo-code for the insertion of elements in the scheduler data structure. When a new element $\langle LP_x, ts\rangle$ needs to be registered into the data structure (indicating that LP_x has become ready for execution of an event with

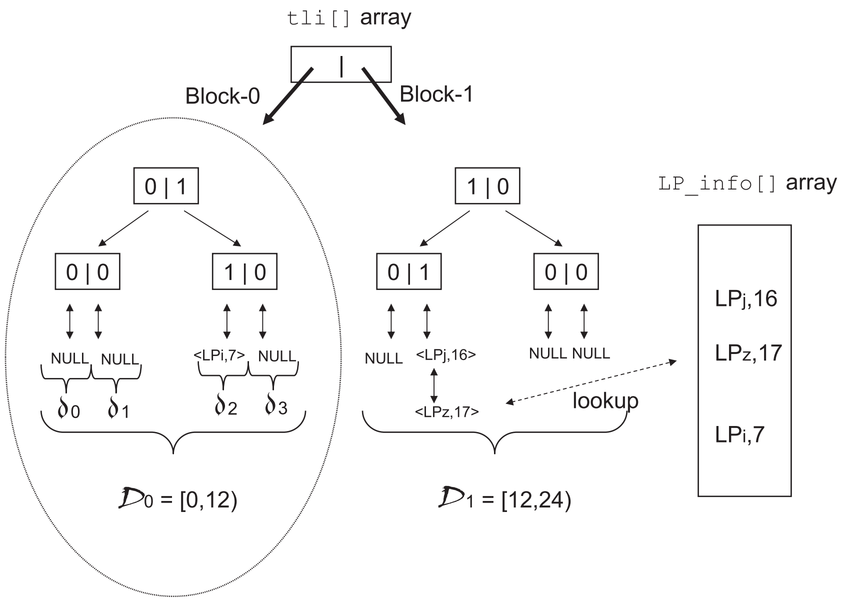

Fig. 2. The tree-like data structure.

timestamp *ts*), it is first hashed, using its timestamp *ts* as the hash key, to determine the time interval Δ_j and the time bucket δ_i within which it falls. This bucket uniquely corresponds to the *i*-th leaf within the *j*-th block of the scheduler data structure (see line 4 and line 6). Then, the element is inserted in the list associated with that leaf, by directly accessing the appropriate tree via the top level reference array `tli[]` (see lines 7–8). Also, in case the corresponding leaf already keeps other elements, a simple (non-ordered) head-insertion within that list is performed. The case of more than one element being registered within the list associated with a single leaf will be referred to as *collision*. Clearly, the list of colliding elements associated with any leaf is not sorted by the timestamps of the elements. Then, the hierarchical bitmap nodes associated with the corresponding branch within the tree are checked, and possibly updated, in order to provide a correct representation of the presence of such a new element within the tree-like data structure (see lines 9–16).

Upon element insertion, the `min_index` variable is updated as follows (see line 17). If an element is inserted within the tree-like data structure having timestamp falling within the interval of simulation time Δ_j associated with the current minimum (i.e. `min_index` is currently set to *j*) or in a subsequent interval, then `min_index` is left unchanged. In case the insertion falls within an interval Δ_k preceding the one associated with the current minimum, `min_index` is simply set to the index *k* of the block corresponding to the interval Δ_k. Finally, all the fields within the `LP_info[x]` entry are updated in order to correctly store the timestamp of the inserted element, and the memory address of the corresponding buffer (see lines 18–19).

Algorithm 1. Insertion of an element within the scheduler data structure.

```
1.  insert $\langle LP_x, ts\rangle$ do
2.    $elem$ = allocate(sizeof($scheduler\_elem$));
3.    set $elem \rightarrow value = \langle LP_x, ts\rangle$;
4.    $j = \lfloor \frac{ts}{\Delta} \rfloor$;
5.    $current\_tree = \&tli[j]$;
6.    $i = \lfloor \frac{ts\%\Delta}{\delta} \rfloor$;
7.    $current\_leaf = current\_tree \rightarrow leaves[i]$;
8.    head_insert $elem$ into $current\_leaf \rightarrow elem\_set$;
9.    $bitmap\_node = current\_leaf \rightarrow bitmap$;
          //this is the reference bitmap node for the current leaf
10.    set $pos = i\%BITMAP\_NODE\_SIZE$;
          //position of the bit associated in the bitmap node
          // with the current leaf
11.    do
12.      if($bitmap\_node[i] == 1$) break;
          //path towards the root of the tree already set as non-empty
13.      $bitmap\_node[i] = 1$;
14.      set $i$ = index($bitmap\_node$);
          //position of the bit associated with this bitmap node
          //in its parent, if any
15.      $bitmap\_node = bitmap\_node \rightarrow parent$;
          //actual parent bitmap node – NULL in case we are already on the root
16.    while($bitmap\_node \neq NULL$);
17.    if($j < min\_index$ or $min\_index$ is not set) $min\_index = j$;
18.    $LP\_info[x].timestamp = ts$;
19.    $LP\_info[x].elem = elem$;
20.  end do
```

4.3. Element extractions

Upon element extraction we have instead the following update rules. For an extraction associated with an LP schedule operation, whose pseudo-code is provided in Algorithm 2, the extracted element (according to LTF) is the one with the current minimum timestamp within the whole data structure. By using `min_index`, the corresponding block is determined, and its hierarchical bitmap nodes are used to determine the time bucket δ_j, and hence the leaf, currently keeping the element with the minimum timestamp (see lines 3–11). In case of collisions on that leaf, a linear search is performed within the corresponding non-ordered list to determine the element with the actual minimum (see line 12). Note that the linear search in the list associated with bucket δ_j is performed (if requested) one time for each schedule operation. Then the target element is removed, and the bitmap nodes within the hierarchical data structure associated with the tree-like block are updated, if actually requested (see lines 14–16). This operation takes place by relying on the **reshuffle** function presented in Algorithm 3. This function also checks if the block indexed by `min_index` becomes empty after the element extraction (see lines 8–14 in Algorithm 3), in which case the top level referencing array is scanned for accessing the root bitmap node associated with subsequent blocks in order to identify the next non-empty block, if any. Then `min_index` is set to the index of the array entry pointing to that block. This operation is similar to what happens with the Calendar-Queue algorithm [11], where a scan across an array of time buckets is operated in similar situations. However, our approach is expected to exhibit reduced overhead since our scan operation checks at each step the presence of at least one element over $N = |\Delta_i|/|\delta_j|$ time buckets, which is achieved thanks to the presence of the hierarchical bitmaps within the scheduler data structure.

Algorithm 2. The schedule operation (extraction of the element with minimum timestamp).

```
1.  schedule void do
2.    if(min_index is not set) return NO_ELEM;
3.    current_tree = &tli[min_ndex];
4.    bitmap_node = current_tree → root_bitmap;
5.    i = 0;
6.    while(i < TREE_DEPTH) do
7.      set pos = most_significant_set_bit(bitmap_node);
8.      bitmap_node = bitmap_node → childs[pos];
          //going onto the child bitmap along the highest priority path
9.      i++;
10.    end do;
11.    current_leaf = bitmap_node → leaves[pos];
          //leaf whose list contains the element with minimum timestamp
12.    find elem ∈ current_leaf → elem_set with minimum timestamp;
          //list scan required in case of collisions
13.    remove elem from current_leaf → elem_set;
14.    bitmap_node = current_leaf → bitmap;
          //this is the reference bitmap node for the current leaf
15.    if(current_leaf → elem_set == NULL) bitmap_node[pos] = 0;
          //this path towards the root became empty after the extraction
16.    reshuffle(bitmap_node);
17.    unset LP_info[elem → LP].timestamp;
18.    unset LP_info[elem → LP].elem;
19.    return elem;
20.  end do
```

Algorithm 3. The reshuffle operation.

```
1.  reshuffle bitmap* bitmap_node do
2.    do
3.      if(*bitmap_node ≠ 0) break;
          //paths are still valid towards this node – no reshuffle of upper level bitmaps
4.      set pos = index(bitmap_node);
          //position of the specific bit associated with this bitmap node in its parent, if any
5.      if (bitmap_node → parent ≠ NULL) bitmap_node → parent[pos] = 0;
6.      bitmap_node = bitmap_node → parent;
          //actual parent bitmap node – NULL if we are already on the root
7.    while(bitmap_node ≠ NULL);
8.    while(tli[min_index] → root_bitmap == 0) do
9.      if (min_index == TREE_LIKE_BLOCKS)
10.       unset min_index;
          //no other element is currently registered within the scheduler data structure,
          //in any tree-like block
11.         break;
12.       endif;
13.       min_index++;
14.    end do;
15.  end do
```

Algorithm 4. The LP unregister operation.

```
1.  unregister LP_identifier x do
2.    if ($LP\_info[x].timestamp == NO\_VALUE$) return;
          //the LP is not currently registered as ready
3.    set $ts = LP\_info[x].timestamp$;
4.    $j = \lfloor \frac{ts}{\Delta} \rfloor$;
5.    $current\_tree = \&tli[j]$;
6.    $i = \lfloor \frac{ts\%\Delta}{\delta} \rfloor$;
7.    $current\_leaf = current\_tree \rightarrow leaves[i]$;
8.    remove $LP\_info[x].elem$ from $current\_leaf \rightarrow elem\_set$;
          //constant time operation via direct pointer based access to elem
9.    deallocate $LP\_info[x].elem$;
10.     if $current\_leaf \rightarrow elem\_set \neq NULL$ return;
          //no need to reshuffle hierarchical bitmaps, the bucket still keeps elements
11.     $bitmap\_node = current\_leaf \rightarrow bitmap$;
          //this is the reference bitmap node for the current leaf
12.     set $pos$ = index($current\_leaf$);
          //position of the specific bit associated with the current leaf in its reference bitmap
13.     if($current\_leaf \rightarrow elem\_set == NULL$) $bitmap\_node[pos] = 0$;
          //this path towards the root became empty after the extraction
14.     reshuffle($bitmap\_node$);
15.     unset $LP\_info[x].timestamp$;
16.     unset $LP\_info[x].elem$;
17.   end do
```

For an extraction associated with a change of the timestamp of the next event of a given LP, and hence a change of the priority of that LP, we have to perform two different tasks: (A) the element currently registered within the tree-like data structure for that LP needs to be removed (hence the LP needs to be deregistered from the scheduler); (B) the new element must be inserted (leading to registering the LP again as ready for event processing). Overall, the change of priority can be seen as the sequential combination of the two different operations related to tasks A and B.

As for task A, indicating with x the identifier of the LP for which the removal must take place, the operation can be executed by directly accessing the element to be removed via the `LP_info[x].elem` reference, and then rearranging the pointers to the next and preceding elements within the corresponding double-linked list. The removal might also require updating the hierarchical bitmap nodes associated with the representation of the current state of the branch toward that leaf, as well as the `LP_info[]` array and `min_index`. The pseudo-code for a removal is shown in Algorithm 4. It relies on the already presented **reshuffle** function in Algorithm 3 to carry out the update of the hierarchical bitmap nodes and of the value of `min_index`. On the other hand, task B gets performed as an element insertion operation (which has been already discussed in Section 4.2).

We finally note that, in optimistic simulation systems, a change of the priority of an LP may occur in case the event of that LP currently registered within the tree-like data structure needs to be cancelled due to the arrival of the corresponding anti-event in a rollback phase (which is different from the case where the notification of a new event for the LP takes place, whose timestamp is lower that the one currently registered within the scheduler). In case the cancellation leads the target LP to become no more ready for event processing (due to absence of additional pending events in its future-event-list), then the whole process boils down to a simple removal involving operations related to task A only (namely, Algorithm 4).

4.4. Time complexity

In this section we provide hints on the time complexity of Algorithms 1–4. As a preliminary note, the parameters determining the arity, denoted as *TREE_ARITY*, and depth, denoted as *TREE_DEPTH*, of the tree-like data structure are fixed, and independent of the number of elements registered within the scheduler data structure.

We assume that the number of collisions occurring within the list associated with any bucket δ_i is upper bounded by a constant value we denote as *MAX_COLLISIONS*. The number of tree-like blocks, denoted as *TREE_LIKE_BLOCKS* in the pseudo-code, is also assumed to be a configuration parameter, independent of the total number of elements kept by the scheduler data structure. How to dynamically resize the tree-like data structure, while still keeping *TREE_ARITY*, *TREE_DEPTH* and *MAX_COLLISIONS* fixed and *TREE_LIKE_BLOCKS* upped bounded by a maximum value independent of the number of elements kept by the scheduler data structure, will be discussed in Section 4.6.

As a last preliminary note, being the value of *TREE_ARITY* fixed, any operation of retrieving the most significant bit within bitmap-nodes (see, e.g., line 7 of Algorithm 2) has constant time, given that the number of bits to check is independent of the number of elements registered within the scheduler data structure (it only depends on the fixed *TREE_ARITY* parameter). Therefore, any operation involving most significant set-bit search (or bit position identification) within bitmap nodes is implicitly assumed to have constant time, and not further analyzed.

4.4.1. Analysis of Algorithm 1

In Algorithm 1, all the statements except the ones in lines 11–16 are simple assignments (or conditional ones). Hence their time complexity is independent of the number of elements kept by the scheduler data structure. They can be therefore executed in $O(1)$ time.

On the other hand, the cyclic block of statements in lines 11–16 has as control information the current content of *bitmap_node* (in fact the cycle ends when *bitmap_node*$[i]$ is found to be non-zero or *bitmap_node* is found to be *NULL*). Given that:

- this control information is set to correspond to *current_leaf* $\rightarrow$ *bitmap* when the cyclic block is started;
- at each iteration *bitmap_node* is updated in order to represent the parent bitmap in the tree-like data structure (particularly in one block of this data structure);
- the tree-like root has parent bitmap-node reference set to *NULL*;

we have that the maximum number of times the cyclic block in lines 11–16 is executed corresponds to the depth *TREE_DEPTH* of the tree-like data structure. Hence the time complexity for this cycle is $O(TREE\_DEPTH)$. However, being *TREE_DEPTH* constant and independent of the number of elements registered within the data structure, we get that the actual complexity of the cyclic block is $O(1)$. As a consequence, the time complexity of the whole set of statements in Algorithm 1 is $O(1)$, hence this algorithm can be executed in constant time vs the number of elements kept by the scheduler data structure.

4.4.2. Analysis of Algorithm 2

In Algorithm 2, all the statements except the cyclic block in lines 6–10, plus the list search in line 12, are simple (or conditional) assignments or function calls (see line 16, where a call is issued to the function **reshuffle** presented in Algorithm 3, whose complexity will be analyzed in the next paragraph). Hence their cumulative complexity is $O(1)$. Therefore we focus our analysis on the above mentioned cyclic and list search operations:

- the cycle in lines 6–10 has control variable i initially set to zero, and incremented by one at each iteration. Also, the ending condition of the cyclic block is $i \geqslant TREE\_DEPTH$, hence the cycle is execute at most *TREE_DEPTH* times, giving rise to $O(TREE\_DEPTH)$ complexity. However, being *TREE_DEPTH* a constant value independent of the number of elements kept by the scheduler data structure the actual complexity of this cyclic block is $O(1)$;
- the list search in line 12 is used to retrieve the element with minimum timestamp in the list associated with an individual bucked, namely *current_leaf*. By having this list containing at most a number of elements equal to *MAX_COLLISIONS*, the time complexity for the linear search operation is $O(MAX\_COLLISIONS)$. However, the maximum number *MAX_COLLISIONS* of elements kept in the list (being fixed) is independent of the total number of elements kept by the whole tree-like data structure. Hence, the list scan complexity has $O(1)$ actual complexity.

By the above arguments, the time complexity for the whole set of statements in Algorithm 2 is $O(1)$, hence this algorithm can be executed in constant time.

4.4.3. Analysis of Algorithm 3

Algorithm 3, which implements the **reshuffle** function called by both Algorithms 2 and 4, is made up by two cyclic blocks, one in lines 2–7 and another one in lines 8–14. Both of them are analyzed below:

- as for the cyclic block in lines 2–7, we can apply the same analysis we used before in Algorithm 1, thus getting $O(1)$ for its time complexity;
- the cyclic block in lines 8–14 has control variable *min_index* and break condition *min_index* $==$ *TREE_LIKE_BLOCKS*, leading to time complexity $O(TREE\_LIKE\_BLOCKS)$. However, being the parameter *TREE_LIKE_BLOCKS* independent of the number of elements kept by the scheduler data structure, the actual complexity of this cyclic block is $O(1)$.

Overall, Algorithm 3 has $O(1)$ time complexity, hence it can be executed in constant time.

4.4.4. Analysis of Algorithm 4

In Algorithm 4, all the statements are simple (or conditional) assignments or function calls (see line 14). Hence their time complexity is independent of the number of elements kept by the scheduler data structure. They can be therefore executed in constant time. On the other hand, the unique function called by this algorithm is the **reshuffle** function in Algorithm 3, which has already be shown to have $O(1)$ time complexity. As a consequence, the overall time complexity of Algorithm 4 is $O(1)$.

4.5. Handling overflows

At any point in time, the tree-like data structure covers an interval of simulation time of size $\Delta ST = \Delta \times K$, where K is the current size of the top level indexing array `tli[]` (it corresponds to *TREE_LIKE_BLOCKS* in the pseudo-code). Given the finite size of this data structure, it is possible that the next event timestamp, to be registered by the scheduler for some LP, falls beyond the covered interval. This overflow condition is managed in our proposal by extending the `LP_info[]` lookup array. In particular, each entry of this array has been augmented with a flag indicating whether some valid element for the LP is currently registered within the overflow region (therefore not being present in the tree-like data structure but only within the lookup table). The scheduler also keeps a counter indicating the number of LPs for which the corresponding elements are registered within the normal (non-overflowed) interval. If this number goes to zero, then the extraction of the element with the minimum timestamp according to the LTF rule boils down to a linear-cost search operation of such a minimum value within the overflow records kept by the `LP_info[]` lookup table. This is a case in which non-constant-time is experienced.

However, as noted by various studies, there is typically a strong locality of not yet executed events close to the current GVT value [9,10]. Therefore, by reusing the entries of the `tli[]` array, and the corresponding tree-like blocks, according to a circular buffer policy when the GVT is computed and memory recovery is performed, the likelihood of actual scheduling performed by linear scan of the lookup table should have non-relevant statistical significance in general settings. Clearly, such a statistical significance is also related to the frequency according to which the new GVT value is computed, given that the longer the wall-clock-time since the last GVT computation, the lower the actual locality of events around that value of the GVT (in fact the last computed GVT value is a lower bound approximation for the current GVT). This aspect will be further discussed on Section 4.6, where we cope with the dynamic re-size of the scheduler data structure in relation to the dynamics of the specific optimistic simulation run.

To cope with re-usage of the entries of the `tli[]` array, and of the tree-like blocks, the actual implementation of our scheduler offers an apposite API that can be invoked by the simulation platform after a newly computed GVT value is available. Upon invocation of this API, every block associated with an interval of simulation time Δ_i whose upper extreme is lower than the computed GVT value is recovered according to the circular buffer policy since it is guaranteed that no more element will eventually have timestamp falling within that interval (where simulation execution is already committed).

4.6. Run time optimization of the scheduler parameters

As we pointed out, the scheduler exhibits constant-time when scheduling operations refer to elements located within the non-overflowed simulation time region, covered by the blocks registered within `tli[]`, and when the number of collisions within each bucket δ_j is bounded by the constant value *MAX_COLLISIONS*. Therefore, depending on the application-specific event pattern, and on the density of the events along the simulation-time axis, as well as on the frequency of calculation of the GVT value, we need to dynamically alter the parameters defining the actual size of the scheduler data structures and the granularity of the buckets in order to maintain the property that a (e.g. compile-time defined) fraction of scheduling operations must refer to the non-overflowed region, and must take place over a bucket-list comprising at most *MAX_COLLISIONS* elements.

To achieve this objective, once fixed the depth and arity of the tree-like blocks, we have devised an approach where starting from some default values for the number of entries of the `tli[]` array, and for the size δ of any bucket, these values are dynamically updated depending on run-time statistics (as typical of approaches such as the Calendar-Queue, where dynamic resize is envisioned [11]).

In our proposal, the rules for reconfiguring the scheduler periodically, along subsequent observation windows, are the following ones:

R1 – Bucket compression:
in case the observed average number of collisions C while performing schedule operations oversteps the fixed value *MAX_COLLISIONS* ([4]), the virtual time interval covered by the set of tree-like blocks is compressed by the factor $\frac{MAX\_COLLISIONS}{C}$. This is achieved by shrinking the length of the virtual-time interval δ (associated with any generic δ_i bucket) by that same factor. However, in order to avoid excessive compression due to spike-values for the event density along the simulation-time period the observations in the last window refer to, we rely on a threshold value TR, such that $0 \leqslant TR \leqslant 1$, to be used for calculating the compression factor as $max(\frac{MAX\_COLLISIONS}{C},\ 1 - TR)$. We note that this scaling operation does not require resizing the scheduler data structure, given that the number of entries of the `tli[]` array (and hence the number of tree-like blocks) remains unchanged.

R2 – Non-overflowed region stretching:
in case the percentage P of the schedule operations falling outside the virtual-time interval covered by the tree-like blocks of the scheduler exceeds a threshold percentage TP (recall that this kind of operations require linear scan of

[4] The value of C is computed as the average number of elements to be scanned in the target-bucket list when performing a schedule operation, namely the extraction of the element with the minimum timestamp.

the `LP_info[]` array, given that all the LPs ready for dispatching are registered in the overflow-zone, which leads to non-constant-time scheduling), the virtual time interval covered by the whole set of tree-like blocks is stretched by increasing the number of entries of the `tli[]` array, and consequently the number of tree-like scheduler blocks, by a factor $(1+P)$. The value of P is computed as the inverse-ratio between the total number of schedule operations and those for which the tree-like data structure is found to keep no elements (given that all the elements, if any, are registered within the overflow table).

Clearly, the above two rules have relations with each other given that a compression related to **R1** may lead to an increase of the value of P along subsequent observation periods, which in its turn may trigger a stretch operation of the simulation-time interval to be covered by the non-overflowing zone, according to **R2**. Also, theoretically **R2** might give rise to unbounded growth of the amount of memory to be reserved for the scheduler data structures, which is clearly not viable in practice. To cope with this issue, we have complemented the above rules by imposing an upper bound on the whole virtual-memory size for the scheduler tree-like blocks. In case this upper bound is reached, **R1** is actuated only if the value of P does not exceed TP. In fact, actuating bucket compression when $P > TP$ will likely lead the scheduler to work exclusively in the overflow-zone during the subsequent observation periods, thus giving rise to non-constant-time scheduling operations. On the other hand, in case the memory-usage upper bound is reached, **R2** cannot be actuated at all. In such a case, we use a fall-back rule complementary to **R1**, based on stretching the bucket size δ by the factor $(1+P)$. This is expected to reduce the likelihood of linear scan over the whole `LP_info[]` table when performing a schedule operation, at the expense of a potential increase of the actual number of collisions within any bucket beyond the predetermined *MAX_COLLISIONS* value. This is expected to well amortize over-collision scenarios and their impact on performance. Other approaches could be envisioned, such as using multiple lower level lists (associated with fractions of the time bucket interval), each one again bounded by *MAX_COLLISIONS* in size, which is the approach used by the Ladder-Queue proposal [25] in case of highly skewed distribution of the timestamps of the registered elements within the queue ([5]). This approach is anyway orthogonal to the core (both data structures and management logic) of the present proposal.

4.7. Memory layout of the scheduler blocks

As said, our objective is to design a constant-time scheduler which also provides low actual overhead. To this aim, some choices have been made in order to improve the access performance to the scheduler data structure, especially in relation to the target x86 architectures. In particular, in our design the tree associated with each block is represented in memory in a very convenient way, both for its memory layout and the related memory access operations. Let h be the arity of the tree, and d its depth, the tree will have a total number of $(h^d - 1)/(h-1)$ bitmap nodes and h^d leafs. The tree is allocated via a couple of contiguous arrays of the above sizes, respectively. The first array is a contiguous memory chunk storing bitmap nodes, and is organized in a classical packing way of trees onto a linear data structure by letting the root node of the tree be in the first position of the array (index = 0), and by identifying the index of the j-th child of the i-th node as $index(i,j) = i \times h + 1 + j$. By this organization, the operation of rising up along the tree towards the root is immediate. In fact, it is sufficient to make an integer division to obtain the index of the parent of a node. A similar operation can be carried out to find the position of the child relative to its parent, which is especially useful for the purpose of updating the bitmaps. We simply calculate the module of the index of the node minus 1, over h.

The second array, storing the pointers to the lists associated with each block, allows immediate access to the list of interest for both element insertion and extraction. Given the timestamp associated with the element to be inserted/extracted, as already discussed, the hashing mechanism for the identification of the corresponding bucket (and hence of the corresponding entry in this array) is simply implemented as a couple of divisions, first between the timestamp value and Δ, then between the rest of this division and δ.

Bitmap nodes are actually represented in memory as words, so their length is proportional to the maximum word length achievable with the particular architecture the software is run on (it will most likely be either 32 or 64 bits). A trivial way of finding the first set bit within a bitmap node, which is exactly the way followed by the `ffs` library function, would be to bitwise rotate the word itself by one bit at a time, until a bit in a particular position is found to be set. However, this solution relies on an approach requiring to iterate, in the worst case, over all the bits within the bitmap node. Therefore, we opted for an optimized different solution which takes advantage of the floating-point capabilities offered by modern architectures. Specifically, in our implementation, the bitmaps are actually stored in memory with the most-significant-bit referring to child #0, and the least-significant-bit referring to the highest-level child. Also, via assembly blocks nested within the `C` code, we exploited the `fyl2x` instruction offered by the x87 instruction set, which calculates the base 2 log of the argument multiplied by a scalar. By setting the scalar value to 1, a simple base 2-log is calculated, which is the position of the most significant bit set. This value can then be used to traverse the tree by directly accessing the child with the requested index.

[5] As shown by the theoretical analysis in [25], up to 8 indirection levels on bucket lists are envisioned as adequate for coping with most of the timestamp distributions of registered elements.

5. Experimental data

5.1. Simulation platform

All the modules implementing the presented LTF scheduler have been made available for free download within the ROOT-Sim platform, which is an open source C/MPI-based simulation package targeted at POSIX systems [28]. It implements a general-purpose parallel/distributed simulation environment relying on the optimistic synchronization paradigm.

ROOT-Sim offers a very simple programming model based on the classical notion of simulation-event handlers (as typical of several well known PDES systems [6–8]) to be implemented according to the ANSI-C standard, which represent application entry points for providing control to the LPs. Also, it transparently supports all the services required to parallelize the execution and offers a set of optimized protocols aimed at minimizing the run-time overhead, thus allowing for high performance and scalability. Among them we can mention an autonomic protocol for application transparent and performance optimized management of state log/restore operations [29], and the support for load balancing and load sharing [30,31].

Before the LTF scheduler proposed in this article was integrated, the CPU scheduling subsystem in ROOT-Sim relied on an $O(n)$ approach similar to the one provided by Linux 2.4.x kernels. This scheduler version simply accesses the event queues of the LPs to determine, via a sequential scan operation of an array of pointers to the next event of the LPs within the respective queues, the LP to be scheduled according to LTF. The advantage of this approach resides in its simplicity, since the scheduler itself has no internal state (exactly like in Linux 2.4.x, where a unique queue of process control blocks is scanned upon each CPU-schedule operation to calculate the so called *goodness* representing the current priority of a ready process/thread), and simply needs to observe the current state of the event queues of the LPs to determine CPU assignment. However, the integration of the currently proposed low-overhead, constant-time scheduler version allows for highly improved scalability of the scheduler delay vs the size of the overlying simulation model, as we will also demonstrate via experimental data.

As for the configuration of the constant-time LTF scheduler in the experiments, the selected parameter values are reported in Table 1. Also, the reconfiguration rules **R1** and **R2**, and their variants to be applied in case the maximum amount of virtual memory to be destined for the scheduler tree-like blocks is reached, are actuated each time a new GVT calculation takes place. Hence, the observation window for the parameters determining the outcome of the scheduler reconfiguration (such as C and P) is implicitly assumed to correspond to the GVT calculation period. The actual reconfiguration takes place in a fully transparent manner, and is triggered by the same API used for activating the aforementioned housekeeping operations for supporting the circular buffer policy for the `tli[]` array.

5.2. Benchmark application

For this experimental study we have selected as benchmark application the well known *game of life*, for which a port to ROOT-Sim has been realized. In this application, a bi-dimensional space region is divided into cells, each one modeled by an individual LP, which in our implementation represent octagons. We consider these octagons as distributed over a square-grid, where each octagon has 4 physically adjacent neighbors, and other 4 virtually adjacent neighbors corresponding to the octagons located along the 4 polar directions. Hence, except for border cells, each cell has a total of 8 neighbors. At any point along simulation time, any cell can be in one of the following states: *dead* or *alive*. Also, each time a cell switches state, it notifies its new state value to all the neighbors, by scheduling *state notification* events.

Upon processing *state notification* events incoming from any neighbor, a cell can either change its local state or persist into the current state. This occurs according to the following classical rules [32]:

Rule number 1:	Any alive cell with fewer than two alive neighbors dies, as if caused by under-population.
Rule number 2:	Any alive cell with two or three alive neighbors keeps staying alive.
Rule number 3:	Any alive cell with more than three alive neighbors dies, as if by overcrowding.
Rule number 4:	Any dead cell with exactly three alive neighbors becomes a live cell, as if by reproduction.

In our implementation, the reevaluation of the above rules does not take place right upon processing the *state notification* event incoming from some neighbor. Rather, each cell is in charge of scheduling for itself a *clock advance* event, upon the

Table 1
Configuration of the LTF scheduler parameters.

Initial bucket size δ	1 virtual time unit
Initial number of tree-like blocks	10
Maximum number of tree-like blocks	100
Arity of each tree-like block	16
Depth of each three-like block	2
Total number of buckets (leaves) per tree-like block	256
Initial span of the non-overflow region	2560 virtual time units
Value of *MAX_COLLISIONS*	50 (in accordance with [25])
Value of *TR*	0.1
Value of *TP*	0.05

occurrence of which the rules are reevaluated and a change in the local state, if any, is notified to the neighbors. *Clock advance* events are scheduled with timestamps that are calculated by summing the local clock value of the cell to an exponentially distributed virtual time increment, whose average value has been set to 1 simulation time unit. On the other hand, in case upon the occurrence of the *clock advance* event the local state of the cell changes, *state notification* events are scheduled towards the neighbors with minimal timestamp increment, whose average value has been set to 0.01, which again follows the exponential distribution. In the experiments we have varied the number of cells, namely LPs, involved in the run (i.e. the simulation model size) between 10,000 and 100,000.

The above depicted configuration of the *game of life* allows assessing the effectiveness of the proposed LTF scheduler by considering a complex event pattern, in terms of the distribution of the event timestamps along simulation time. Particularly, the presence of *clock advance* events, scheduled on a periodic basis, tends to generate an even distribution of event timestamps along simulation time, with actual density that changes (namely increases) while increasing the size of the simulation model. On the other hand, *state notification* events tend to create peaks in the density of event timestamps along intervals where state updates actually occur, given that the notification of a state update to the neighbors can give rise to cascades of notification events across cells in a given region. This gives rise to peaks of locality of events in specific simulated time intervals, thus generating a highly variable density pattern, also depending on the base density of *clock advance* events, which, as already hinted, changes vs the model size. Further, when the peak materializes, we get non-minimal likelihood that the LPs involved in the notification process will change their priority, given that *state notification* events are generated with very small timestamp increment compared to the periodically scheduled *clock advancement* events. Hence, the spreading of state updates will lead, for the involved LPs, to unregister the already scheduled *clock advancement* event from the scheduler data structures, and to in place register the *state notification* event as the LP next event. This will give rise to executions where all the modules implementing the scheduler logic will play a relevant role, namely the module implementing the scheduling of the highest priority LP (see Algorithm 2) and the ones implementing the unregistering/registering of the LPs within the scheduler (see Algorithms 1 and 4), the latter being used any time the LPs change their CPU-scheduling priority.

5.3. Hardware and system software settings

The hardware architecture used for testing our proposal is a 64 bit NUMA machine, namely an HP ProLiant server, equipped with four 2 GHz AMD Opteron 6128 processors and 64 GB of RAM. Each processor has 8 cores (for a total of 32 cores) that share a 12 MB L3 cache (6 MB per each 4-cores set), and each core has a 512 KB private L2 cache. The Operating System is 64 bit Debian 6, with Linux kernel version 2.6.32.5. The compiling and linking tools used are `gcc` 4.3.4 and binutils (`as` and `ld`) 2.20.0.

32 instances of the ROOT-Sim kernel have been run on this platform (one per CPU-core), with even distribution of the LPs forming the simulation model onto the different kernel instances. Regarding ROOT-Sim run-time parameters, the GVT period (namely, the interval for memory recovery of obsolete logs, at the end of which we also trigger the reconfiguration of the LTF scheduler parameters) has been set to 1.5 s. With this value, RAM usage never exceeded 60%/70%, thus avoiding swapping phenomena that would alter the reliability of the reported measures.

5.4. Results

In the experimental study we have compared the run-time behavior of our low-overhead constant-time LTF scheduler, which we refer to as LOCT in the plots, with other two alternatives. One is a classical $O(n)$ linear-time scheduler, which has been at the heart of ROOT-Sim CPU scheduling support prior to including LOCT within the software package. This is a baseline (non-optimized) scheduler to be used as a reference for assessing the advantages from optimized schedulers. Details on the behavior of such a linear scheduler have been already provided in Section 5.1. The second tested alternative is a Ladder-Queue scheduler. However, given that the Ladder-Queue has not been designed for removing events with non-minimal timestamp, we have added this facility in order to correctly support LTF-style CPU scheduling activities within the ROOT-Sim PDES system. As hinted, the removal of an event with non-minimal timestamp may occur in case an LP changes its priority due to the arrival of an event with timestamp lower than the one associated with the LP and currently registered within the Ladder-Queue as its next event, or due to the arrival of an antimessage exactly annihilating that same event or causing a rollback (which ultimately leads to a change of the priority of the LP). In our variant of the Ladder-Queue, the removal of an event with non-minimal timestamp has been supported via a linear search within the time bucket related to the event-timestamp. We note that this boils down to a linear search into a list, given that the ladder-queue keeps the records associated with each bucket as a linked list. We have decided to select the Ladder-Queue in the experimental comparison given the interest that (as already pointed out) recently arose in using it as the CPU scheduling support structure in modern PDES systems (see, e.g., [26]). For the Ladder-Queue scheduler, we again impose that the maximum number of elements within each bucket is upper-bounded by the value $MAX\_COLLISIONS = 50$ (which corresponds to the suggested value in [25]). Further, to ensure fair comparison with LOCT, the initial number of buckets to be held within the Ladder-Queue has been set to the value 256×10, the maximum number of buckets has been set to the value 256×100, and the initial span of each bucket has been set to 1 virtual time units, in compliance with the parameter values listed in Table 1. The resize of the virtual-time coverage by an individual bucket and of the number of buckets has been based on the classical rules provided in [25]. Further, also for the Ladder-Queue implementation the overflow region is instantiated as an array of entries, one per LP.

For all the experimentally investigated parameters, each value we report is the average over 10 runs, where the initial state of the simulation model in each run has been determined by setting a cell as alive or dead depending on the value of a specific bit in a bmp file. The 10 different bmp files associated with the 10 different runs were selected randomly. Also, the average values we report have high statistical significance given that for each investigated configuration (in terms of model size and selected CPU scheduler) the individual samples exhibited distance from each other of (at most) the order of 10%.

In Fig. 3 we report data related to the *event rate* achieved by the different configurations, while varying the model size. We recall that, in optimistic PDES runs, the event rate is computed as the number of committed events per wall-clock-time unit, hence it is a representation of the speed according to which the optimistic parallel run progresses (in terms of useful work done, namely processed events that are not eventually rolled back). For completeness of the analysis we also report the event rate achieved by the sequential execution of the same identical application code run on top of the Ladder-Queue scheduler ([6]). Here any processed event is also a committed one, given that no out of timestamp-order can ever occur in the sequential run. This curve serves to determine whether (and at what extent) the parallel runs provide speedup, so as to assess whether the comparative analysis of the different schedulers for optimistic PDES systems has been carried out in scenarios with competitive parallel executions. This is the actual case, given that the parallel configurations with LOCT and the Ladder-Queue schedulers both provide significant speedup over the sequential run independently of the model size. By the data we observe how both LOCT and the linear scheduler show a trend where the event rate decreases while the model size is increased. However, LOCT exhibits order of magnitudes better event rate values. In fact, as expected, the linear scheduler shows excessive relative cost for scheduling operations, compared to the cost for processing an event, which leads the event rate to asymptotically tend to the value zero when increasing the model size. This is confirmed by the data reported in Fig. 4, where we show the average cost for an individual (per processed event) CPU-schedule operation in the different configurations. For the case of stateful schedulers, such as LOCT and the Ladder-Queue based one, the average scheduling cost per event reported in the plots also accounts for the time spent while performing management operations on the scheduler data structures (such as resize or clean operations taking place upon GVT computation) and while changing the priority of any LP (which requires extracting the element currently associated with the LP in the scheduler data structures, and inserting a different one). In other words, we report the per executed event overhead by the CPU scheduling subsystem considering the cost spent while executing any function implementing the scheduler logic. By these data we can see how the linear scheduler provides acceptable per event CPU scheduling delays, comparable to the ones provided by the Ladder-Queue scheduler, only for smaller model size values, while it causes a drastic increase of the average per event CPU-schedule time as soon as the model size is increased. Also, as expected, such an increment is linear vs the model size. On the other hand, LOCT provides per event average scheduling cost which is definitely lower than the ones provided by the other two schedulers. This is one of the reasons leading LOCT to also provide event rate values that are better that the ones provided by the Ladder-Queue scheduler. Particularly, LOCT shows a minimum gain in the event rate which is on the order of 16–23%, as observed for larger model sizes (e.g. ranging from 60,000/70,000 to 100,000 cells). On the other hand, the gain in the event rate provided by LOCT over the Ladder-Queue scheduler is definitely greater when considering smaller model sizes. Particularly, the maximum gain, which is on the order of 250%, is noted for model size set to 10,000 cells. This gain is not due to significant changes on the rollback pattern when considering the different schedulers. In fact, as shown in Fig. 5, LOCT and the Ladder-Queue schedulers exhibit quite close efficiency values ([7]) for all the considered model sizes, with the Ladder-Queue scheduler providing slightly better values of the efficiency, which would even tend to favor performance.

The actual motivation for the very large gain in performance provided by LOCT can be explained by looking at the data we report in Fig. 6, where we show the average number of cache-misses (specifically LLC misses) per committed event we have observed with the three different schedulers. This value has been computed by measuring the total number of cache-misses experienced during the run ([8]), which includes cache-misses caused by any memory access while running the simulation platform (thus not limiting the count to accesses caused by the actual processing of the events at the LPs). This parameter is an indicator of the execution locality of the simulation platform as a whole in the different configurations, given that it expresses the level of efficiency according to which the simulation platform exploits the cache for carrying out the operations required for processing any useful (not eventually rolled back) event, namely any unit of productive simulation work. By the data we see how the linear scheduler tends to degrade locality while the model size is increased, which is due to the cache-adverse pattern generated by the scan of scattered data structures (the top element of the future-event-list of each LP), whose amount increases while increasing the model size, to be carried out upon each CPU-schedule operation. Such a scan exhibits very poor locality, which tends to degrade the efficiency of the caching system in relation to all the operations executed by the simulation platform. This is one of the causes for the significant increase of the per event scheduling cost we have already commented on (see Fig. 4). On the other hand, we can see how for reduced model size values, the Ladder-Queue scheduler does not favor locality, in fact it exhibits an average number of cache-misses per committed event which is 240% greater than the corresponding value

[6] Differently from the case of parallel execution, in the sequential run the Ladder-Queue is used, as usual, to register all the scheduled events, not only the next-to-be-processed events of the LPs.

[7] The *efficiency* of an optimistic PDES run is computed as the ratio between the number committed of events and the total number of executed events, namely committed plus rolled back.

[8] This parameter has been measured by relying on the perf tool supported by Linux.

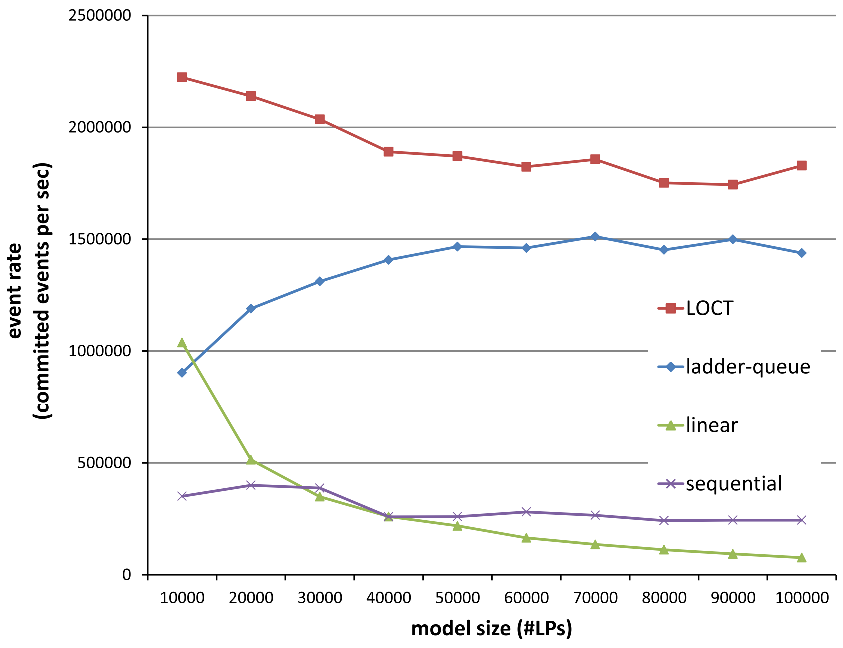

Fig. 3. Event rate.

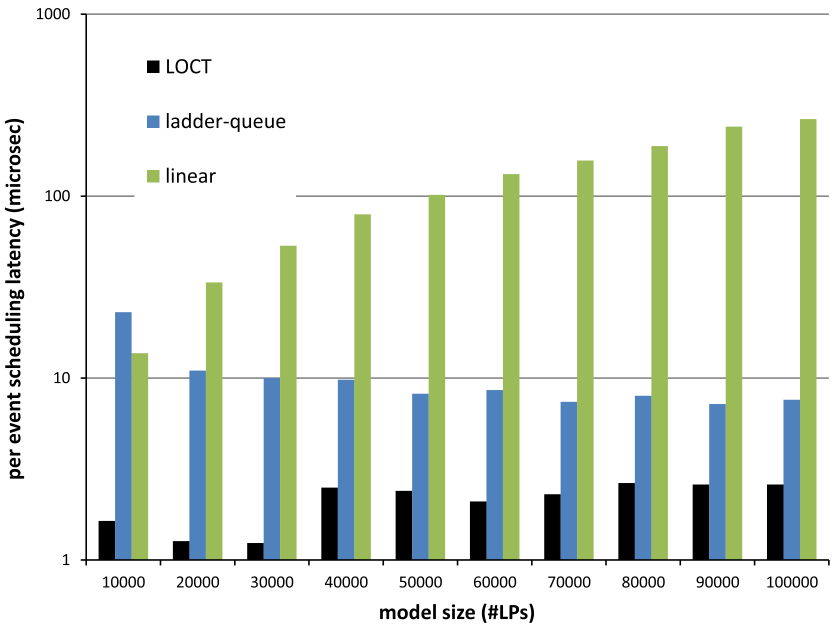

Fig. 4. Per event CPU-scheduling cost (log scale on the *y*-axis).

provided by LOCT. Better locality by LOCT comes out by (i) the reliance on the hierarchical bitmaps, which avoids the need for per-bucket scanning operations that are instead carried out by the Ladder-Queue scheduler in order to find non-empty buckets when scheduling tasks (i.e., min-element extractions) are performed and (ii) the reliance on the `LP_info[]` lookup table, which allows for changing the priority of any LP in constant-time, without the need for scanning the list associated with the bucket keeping the element representing the current (to be changed) priority of the LP, operation which is instead required by the Ladder-Queue scheduler. Such a significant reduction of locality by the Ladder-Queue scheduler leads LOCT to provide the aforementioned noticeable gain in the event rate. On the other hand, when the model size is increased, the level of locality naturally decreases, thus leading the impact on locality by the manipulation of the scheduler data structures less relevant. Hence both the Ladder-Queue and LOCT tend to exhibit similar values of the per event cache-miss count. However, LOCT still provides

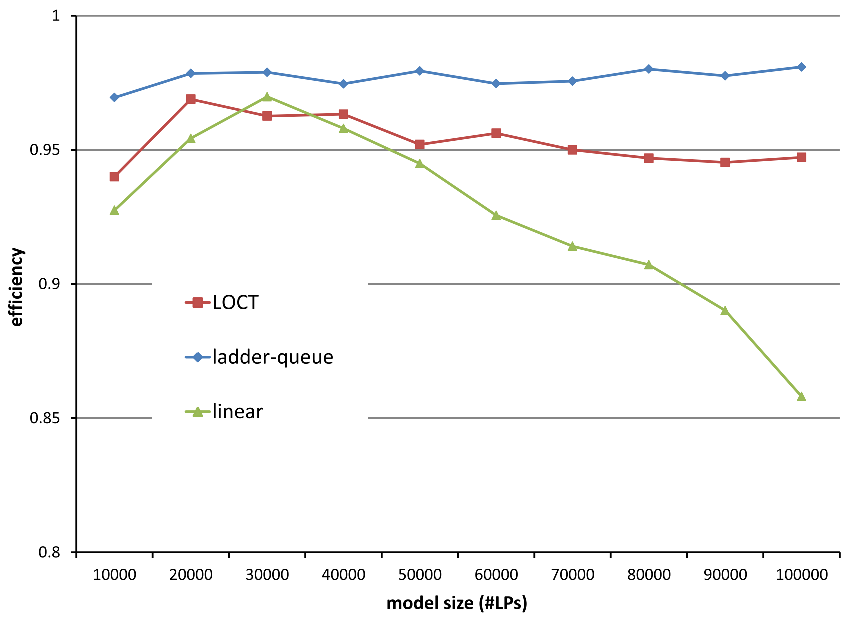

Fig. 5. Efficiency.

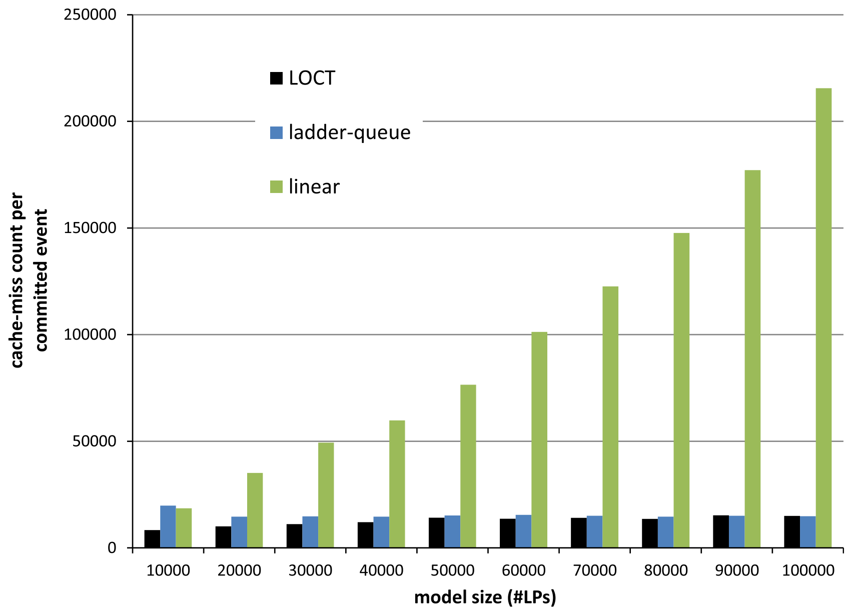

Fig. 6. Cache-miss count per committed event.

performance gains due to the avoidance of the aforementioned scan operations, which favorably impacts the actual overhead for CPU scheduling. Overall, for smaller model sizes, LOCT can favor performance also via secondary effects related to locality improvements. Further, the actual per-event CPU-scheduling cost stays flat while increasing the model size (e.g. beyond 40,000 cells), as shown in Fig. 4. Hence, the slight decrease in the event rate for definitely large model sizes when LOCT is used (e.g. beyond 20,000 LPs) is imputable to the slight increase of the amount of rollback in the parallel run (see Fig. 5), which tends to amplify the overhead for recoverability tasks and to reduce the percentage of useful work carried out per wall-clock-time unit.

To further complement the above results, we report in Fig. 7 the average number of buckets requested by LOCT and by the Ladder-Queue scheduler in order to match the constraint of having the number of collisions bounded by the value *MAX_COLLISIONS*, which has been set to 50 in our study. To compute such an average value without biasing it towards

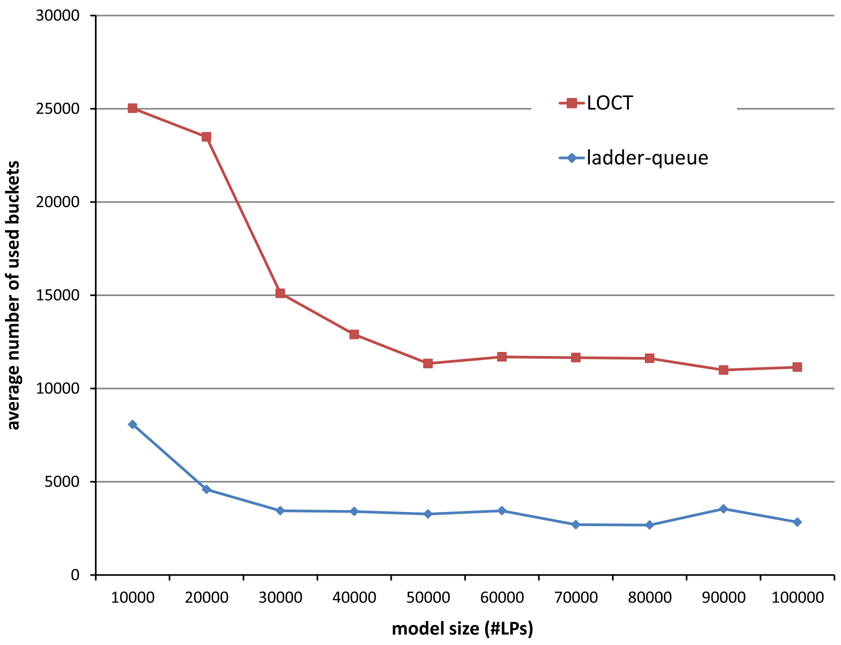

Fig. 7. Average number of used buckets.

the initially selected number of buckets, namely 256 $\times$ 10 – see Table 1 – we have discarded the first 5 samples, namely the number of buckets selected by the schedulers up to the fifth GVT computation and associated resize of the scheduler data structures. By the data we see how LOCT requires more buckets to be held in order to match the constraint, especially for smaller model sizes. This is a reflection of the fact that LOCT allows the simulation run to progress (much) faster than what allowed by the Ladder-Queue scheduler (especially for smaller models), which leads to the need for spanning scheduling activities over a stretched virtual time period (achieved via rule **R2** as specified in Section 4.6, which leads to increased the number of buckets to be managed by the scheduler) prior the calculation of a new GVT value occurs and the scheduler buckets can be recovered (namely fossil collected) for a subsequent execution phase ([9]). However, by the already discussed data provided in Fig. 6, the larger number of buckets to be kept active between subsequent GVT computations does not unfavor cache efficiency when LOCT is employed, which means that these buckets are accessed with pretty good locality, which is an expected result when considering the role of the hierarchical bitmaps that are used by LOCT just to compactly represent (and query) the current state of each bucket (empty or not). In fact, with the selected arity of 16 for the tree-like data structure in LOCT in this experimental study, an empty tree-like block (containing 256 buckets in our experimental settings) can be queried (and hence discovered as empty) by accessing a single 16-bit mask rather than a set of 256 flags, each one explicitly representing the state of an individual bucket. On the other hand, longer scan operations are just requested by the Ladder-Queue scheduler in case of multiple subsequent empty buckets, which leads LOCT to scale down by a factor (up to) 16 the actual number of scan operations requested in the same scenario, which again contributes to favor locality when LOCT is employed (we already provided hints on this aspect while discussing the caching efficiency provided by the Ladder-Queue scheduler – see Fig. 6 – and its relation with the additional per-bucket scan operations requested by this scheduler).

As a last note, the efficiency observed with the linear scheduler (see Fig. 5) initially tends to increase and then to significantly decrease while increasing the model size. This is a reflection of the increase of the per-event scheduling delay. In fact, literature studies (e.g. [18]) have shown how optimistic PDES systems can initially exhibit throttling phenomena when some housekeeping operation takes slightly longer time, which tends to reduce the amount of rollback. However, these systems tend to suffer from trashing phenomena, leading to an increase of the amount of rollback in the parallel run, in case specific housekeeping operations (such as the CPU scheduling operation in our context) induce an excessive delay in the processing of the events. This is exactly the case for the linear scheduler, since significantly longer latencies in the CPU scheduling phase, experienced for larger model size values, lead to higher skewness in the punctual advancement of the local clocks of the different LPs, leading to a noticeable increase in the incidence of rollbacks. This phenomenon is avoided when relying on much more performance effective schedulers such as LOCT.

[9] As pointed out in Section 4.5, fossil collecting the buckets does not imply de-allocating them since the tree-like blocks are used according to a circular buffer policy. Overall, fossil collection simply leads to reusing these buckets binding them to a successive virtual time interval.

6. Conclusions

Parallel Discrete Event Simulation (PDES) is a classical technique for exploiting the computing power of parallel/distributed platforms in order to speedup the execution of simulation models and to make very large models tractable. In PDES platforms, multiple processes/threads are in charge of concurrently running interacting simulation objects (each one modeling a portion of the entire system to be simulated) and one major issue to cope with is related to selecting the next-to-be run object along any worker thread. In PDES systems, all the events, including those resulting from interactions across the concurrent simulation objects, need to be processed in non-decreasing timestamp order by any object for correctness purposes. Optimistic PDES systems are based on speculative processing, hence allowing large exploitation of parallelism, and rely on rollback recovery mechanisms for undoing speculative processing actions that are eventually revealed as incorrect. For these systems, traditional approaches, where the simulation objects that are managed by an individual worker thread have all their pending simulation events registered within a single event-queue, are not feasible given that per-object rollback operations may request frequent reshuffling of the whole event-queue. Hence individual event-queues, one per object, each one keeping a reduced amount of elements to be managed, are preferred. On the other hand, this poses the problem of how to determine the highest priority object to be dispatched by the thread, namely the one associated with the pending event with the minimum timestamp across all the objects managed by that thread when relying on the traditional Lowest-Timestamp-First (LTF) scheduling rule.

In this article we have addressed this problem by presenting an optimistic-PDES oriented LTF scheduler providing constant-time average performance and very reduced actual overhead. This proposal can be seen as a variant of the Calendar-Queue (or Ladder-Queue) where the classical bucket-based organization is complemented with additional data structures that are properly suited for handling scenarios where the scheduler needs to only record the relative priorities of the different objects (hence not all the scheduled, and not yet processed, simulation events), which can (frequently) change over time (e.g. due to rollback operations pushing some object back along the simulation-time axis). Particularly, a single element is registered by the scheduler at any time for any ready-to-run object, namely an object having at least one pending simulation event to be processed, like it occurs in traditional Operating System schedulers, which typically manage priorities by simply registering an individual record per each ready thread into a proper data structure. The main difference with these schedulers lies in that our proposal needs to manage an a priori undetermined amount of priority levels (given that the actual priority of an object is determined on the basis of the timestamp of its next-to-be processed event) while constant-time Operating System schedulers typically manage a finite set of pre-determined priority levels. Also, the proposed scheduler relies on optimizations that are x86 specific, although the whole design and the presented data-structures/management-logic are of general relevance, and are independent of the actual technology used for implementing and hosting the optimistic PDES system.

We have also reported the results of an experimental study based on the *game of life* as benchmark application (that has been run on a 32-core HP ProLiant machine), which show the effectiveness of our proposal. Particularly, by the experimental data we have shown how the proposed solution allows both direct reduction of the cost of the CPU-schedule operation, via reduction of the actual amount of operations (such as record-scan operations) requested to manage the scheduler data structures, and indirect reduction of the impact of the scheduler on the overall efficiency of the PDES platform, which is achieved by improving execution locality and cache efficiency. Particularly, our proposal allows up to 2.5x performance improvements compared to the scenario where the Ladder-Queue is used as the scheduling support, a gain that is achieved for small-to-medium model sizes (namely on the order of 10^4 simulation objects), where cache efficiency plays a major role. On the other hand, for larger model sizes (namely on the order of 10^5 simulation objects) our proposal still provides on the order of 25% performance improvement just thanks to its direct effects. The software package implementing the presented scheduler has been made freely available for download, being it integrated into the open source ROOT-Sim optimistic simulation platform.

References

[1] R.M. Fujimoto, Parallel discrete event simulation, Commun. ACM 33 (10) (1990) 30–53.
[2] A. Park, R. Fujimoto, Optimistic parallel simulation over public resource-computing infrastructures and desktop grids, in: Proc. of the 12th IEEE International Symposium on Distributed Simulation and Real Time Applications.
[3] F. Quaglia, V. Cortellessa, On the processor scheduling problem in time warp synchronization, ACM Trans. Model. Comput. Simul. 12 (3) (2002) 143–175.
[4] T.K. Som, R.G. Sargent, A probabilistic event scheduling policy for optimistic parallel discrete event simulation, in: Proc. of the 12th Workshop on Parallel and Distributed Simulation, IEEE Computer Society, 1998, pp. 56–63.
[5] Y.B. Lin, E.D. Lazowska, Processor scheduling for Time Warp parallel simulation, in: Advances in Parallel and Distributed Simulation, 1991, pp. 11–14.
[6] C.D. Carothers, D. Bauer, S. Pearce, ROSS: a high performance modular Time Warp system, in: Proceedings of the 14th Workshop on Parallel and Distributed Simulation, IEEE Computer Society, 2000, pp. 53–60.
[7] S.R. Das, R.M. Fujimoto, K. Panesar, D. Allison, M. Hybinette, GTW: a time warp system for shared memory multiprocessors, in: Proceedings of the 26th Winter Simulation Conference, Society for Computer Simulation Intl, 1994, pp. 1332–1339.
[8] D.E. Martin, T.J. McBrayer, P.A. Wilsey, WARPED: a time warp simulation kernel for analysis and application development, in: Proceedings of the 29th Hawaii International Conference on System Sciences, Software Technology and Architecture, vol.1, IEEE Computer Society, Washington, DC, USA, 1996, p. 383.
[9] A. Ferscha, Probabilistic adaptive direct optimism control in Time Warp, in: Proceedings of the 9th Workshop on Parallel and Distributed Simulation, IEEE Computer Society, 1995, pp. 120–129.
[10] A. Ferscha, J. Luthi, Estimating rollback overhead for optimism control in Time Warp, in: Proceedings of the 28th Annual Simulation Symposium, IEEE Computer Society, 1995, pp. 2–12.

[11] R. Brown, Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem, Commun. ACM 31 (10) (1988) 1220–1227.
[12] D. Cucuzzo, S. D'Alessio, F. Quaglia, P. Romano, A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation, in: Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications, 2007, pp. 227–234.
[13] K.M. Chandy, J. Misra, Distributed simulation: a case study in design and verification of distributed programs, IEEE Trans. Software Eng. SE-S5 (5) (1979) 440–452.
[14] F. Quaglia, R. Baldoni, Exploiting intra-object dependencies in parallel simulation, Inform. Process. Lett. 70 (3) (1999) 119–125.
[15] D.R. Jefferson, Virtual time, ACM Trans. Programm. Lang. Syst. 7 (3) (1985) 404–425.
[16] F. Quaglia, A. Santoro, B. Ciciani, Tuning of the checkpointing and communication library for optimistic simulation on myrinet based NOWs, in: Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, IEEE Computer Society, 2001, pp. 241–248.
[17] F. Quaglia, A. Santoro, Non-blocking checkpointing for optimistic parallel simulation: description and an implementation, IEEE Trans. Parall. Distr. Syst. 14 (6) (2003) 593–610.
[18] B.R. Preiss, W.M. Loucks, D. MacIntyre, Effects of the checkpoint interval on time and space in Time Warp, ACM Trans. Model. Comput. Simul. 4 (3) (1994) 223–253.
[19] H. Soliman, A. Elmaghraby, An analytical model for hybrid checkpointing in Time Warp distributed simulation, IEEE Trans. Parall. Distr. Syst. 9 (10) (1998) 947–951.
[20] C.D. Carothers, K.S. Perumalla, R.M. Fujimoto, Efficient optimistic parallel simulations using reverse computation, ACM Trans. Model. Comput. Simul. 9 (3) (1999) 224–253.
[21] A.C. Palaniswamy, P.A. Wilsey, Scheduling Time Warp processes using adaptive control techniques, in: Proc. of 1994 Winter Simulation Conference, Society for Computer Simulation, 1994, pp. 731–738.
[22] F. Quaglia, A state-based scheduling algorithm for Time Warp synchronization, in: Proc. of the 33rd Annual Simulation Symposium, IEEE Computer Society, 2000, pp. 14–21.
[23] R. Ronngren, R. Ayani, Service oriented scheduling in Time Warp, in: Proc. of 1994 Winter Simulation Conference, Society for Computer Simulation, 1994, pp. 1340–1346.
[24] R. Rönngren, R. Ayani, A comparative study of parallel and sequential priority queue algorithms, ACM Trans. Model. Comput. Simul. 7 (2) (1997) 157–209.
[25] W.T. Tang, R.S.M. Goh, I.L.-J. Thng, Ladder queue: an O(1) priority queue structure for large-scale discrete event simulation, ACM Trans. Model. Comput. Simul. 15 (3) (2005) 175–204.
[26] T. Dickman, S. Gupta, 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, SIGSIM-PADS '13, ACM, New York, NY, USA, 2013, pp. 103–114.
[27] W. Stallings, Operating Systems, Internals and Design Principles, Prentice Hall, 1998.
[28] HPDCS Research Group, ROOT-Sim: The ROme OpTimistic Simulator – v 1.0. <http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/> (october 2012).
[29] R. Vitali, A. Pellegrini, F. Quaglia, Autonomic state management for optimistic simulation platforms, IEEE Trans. Parall. Distr. Syst. 1 (2014) 1, http://dx.doi.org/10.1109/TPDS.2014.2323967 (preprints).
[30] S. Peluso, D. Didona, F. Quaglia, Application transparent migration of simulation objects with generic memory layout, in: Proceedings of the 25th Workshop on Principles of Advanced and Distributed Simulation, IEEE Computer Society, 2011, pp. 169–177.
[31] R. Vitali, A. Pellegrini, F. Quaglia, Load sharing for optimistic parallel simulations on multi core machines, SIGMETRICS Perform. Eval. Rev. 40 (3) (2012) 2–11.
[32] M. Gardner, Mathematical games "The fantastic combinations of John Conway's new solitaire game life, Sci. Am. 223 (2012) 120–123.
[33] S. Jafer, Q. Liu, G.A. Wainer, Synchronization methods in parallel and distributed discrete-event simulation, Simul. Model. Pract. Theory 30 (2013) 54–73.