The ROme OpTimistic Simulator: A Tutorial

Alessandro Pellegrini and Francesco Quaglia

High Performance and Dependable Computing Systems Research Group
DIAG, Sapienza, University of Rome

Abstract. In this tutorial we present the ROme OpTimistic Simulator (ROOT-Sim), a general-purpose Parallel Discrete Event simulation platform built according to the optimistic synchronization scheme, which allows—via the adoption of a simple/reduced API—to implement simulation models via event handlers relying on standard ANSI-C. We present the set of paradigms which ROOT-Sim is built on, and its internal design, along with the offered facilities. We also explain the simulation-model programming paradigm, and give an example of a basic simulation model, which stands as a building block for more complex ones.

1 Introduction

Simulation is an attractive and well-consolidated methodology to study real-world phenomena. It is continuously exploited in a wide set of fields, including physics, biology and business-oriented processes (such as financial prediction or optimized system-configuration selection). For some application contexts, one relevant aspect relates to the timeliness according to which simulation results are provided to end-users or applications, such as when exploiting simulation as a tool supporting time-critical decision making. Hence, performance aspects while delivering simulation output is a core issue to cope with.

In the context of Discrete Event Simulation (DES), high performance has been targeted via the Parallel-DES (PDES) paradigm [1], which allows exploiting the computing power offered by (high-end) parallel/distributed platforms in order to speedup model execution and to make (very) large and/or accurate models tractable. The basic idea underlying PDES is to partition the simulation model into N distinct simulation objects, which are the core of the simulation process from the model writer's point of view. In fact, each object represents a portion of the real world being simulated. The evolution of simulation objects is described by state transitions, driven by a set of logical/mathematical properties. In order to represent real-world interactions, simulation objects communicate with each other by exchanging pieces of information in the form of events.

From a technical point of view, simulation objects are handled by Logical Processes (LP), which undertake the concurrent execution of simulation events. Traditionally, a PDES run entails a number of concurrent LPs, uniquely identified by a numerical code in the range $[0, N-1]$, and the overall simulation model keeps track of the evolution of the simulated world by relying on a global simulation state S, which is partitioned into various LPs' private and disjoint simulation states S_i, so that $S = S_i \bigcup_{i=0}^{N-1} S_i$ and $S_i \cap S_j = \emptyset, \forall i \neq j$.

D. an Mey et al. (Eds.): Euro-Par 2013 Workshops, LNCS 8374, pp. 501–512, 2014.

In PDES, simulation events are timestamped and their execution is impulsive, meaning that there is no notion of time evolution during an event's processing. The current simulation time at each individual LP is known as Local Virtual Time (LVT), and can be expressed in any measure unit (i.e., one LVT unit can represent seconds, hours, or even years, depending on the actual simulation model). This notion of time is opposed to the Wall-Clock Time (WCT), which is the actual notion of time we are used to. Therefore, in one WCT unit, the LVT advancement can be of one or several units, depending on the actual complexity of the simulation model and on the efficiency of the simulation run.

During the execution of an event, other events can be generated, destined to any simulation object in the system, and are associated with a timestamp value which is greater than or equal to the one of the event currently being executed, i.e. during the execution of the event e_x associated with the timestamp t_x, a new event e_y associated with timestamp t_y can be generated and sent to another simulation object, ensuring that $t_y \geq t_x$. Therefore, event generation evolves according to a causality pattern where the present cannot affect the past.

LPs are in charge of delivering simulation events to the hosted simulation objects, via the invocation of proper event handlers. Simulation-kernel instances take care of dispatching events for processing activities across the various LPs, and of managing inter-LP communication. In particular, they handle the LPs' *event queues*, by reflecting the updates due to incoming messages, and determine the best LP to be dispatched in order to optimize specific execution metrics.

At the same time, when concurrently running the LPs on multiple CPU-cores provided by the underlying parallel/distributed platform, synchronization mechanisms are required in order to ensure that the causality pattern is maintained for both event generation and processing at any LP. Although various definitions of causally consistent execution are devised in literature [2–5], the most widely known and exploited causality criterion expresses that model execution is correct if each LP processes its input events in non-decreasing timestamp order.

To maintain causal consistency two main approaches have been proposed: *conservative* and *optimistic*. The former avoids the possibility of a causal violation at all, relying on block-until-safe policies which suspend event-processing activities at an LP until it is ensured that the execution of its next pending event is coherent with logical-time ordering. The latter—which finds its basis in the Time Warp protocol presented in [6]—aims at full exploitation of the parallelism offered by the underlying computing platform (either local or distributed), which is achieved by avoiding the usage of the aforementioned block-until-safe policies, and by adopting speculative processing. With this approach, causality violations can occur upon the delivery of a so-called *straggler message* to any LP involved in the run, which carries a scheduled event associated with a timestamp t_1 lower than the timestamp t_2 of some already-processed event. In such a case, all the events already processed by the recipient LP, having timestamp t in the interval $t_1 < t \leq t_2$ are no longer causally consistent.

Anytime a causality violation is detected, some rollback recovery mechanism needs to be actuated which involves, at the same time, two orthogonal issues:

1. undoing the effects of inconsistent local processing activities on other LPs;
2. restoring the state of the rolling back LP to a still-consistent past snapshot.

The first task is generally supported via the employment of so-called *anti-messages*, which annihilate events scheduled by the rolling back LP (due to the causal inconsistency). An anti-message is therefore a negative copy of a previously sent message, and is used to signal the destination LP to discard the original message. Clearly, if the event carried by the original message was already processed, we experience a spreading of the rollback occurrence along a chain of LPs, a phenomenon called *cascading rollback*.

On the other hand, the second task poses problems on the side of both performance and application transparency. As for performance, we must consider both CPU usage for supporting checkpoint/restore tasks, and memory usage for keeping recoverability-related data/metadata. On this side, a wide literature exists that has proposed the employment of log/restore techniques (see, e.g., [7–9]), where snapshots of LPs' states are taken according to some (infrequent) policy in order to optimize the tradeoff between log costs and restore latency. Also, these techniques deal either with non-incremental or incremental logging, the latter approach (see, e.g., [10–12]) also trying to reduce the memory usage for log buffers. On the other hand, transparency issues deal with supporting log/restore with no need for state recoverability modules implemented in the application-level code (hence masking the actual synchronization paradigm to the application programmer). This is a non-trivial aspect since it relates to the flexibility according to which the application programmer is allowed to organize the data structures representing the LP state image.

Recently, the memory demand problem associated with logging has been also tackled via *reverse computing* [13], where a state-restore operation is supported via application-level compensation logic that applies backward computation steps until the correct snapshot for resuming the LP is built. For this approach we still find issues in relation to how to generate the reverse code version transparently to the application programmer.

Still in relation to memory usage and recovery, optimistic synchronization is intrinsically linked to the notion of *Global Virtual Time* (GVT). It represents the commit horizon of the optimistic simulation run, i.e. the simulation-time barrier which separates the set of committed events from the still-rollbackable ones. It corresponds to the minimum timestamp of not-yet-processed or in-transit messages/anti-messages. Once the new GVT is available, all the memory buffers keeping events belonging to the committed portion of the simulation (and the related logs) can be released[1]. This procedure is usually termed *fossil collection*. We note that the GVT protocol cannot be executed with unbounded frequency since it imposes some overhead. This leads to further exacerbation of the memory-demand problem, since memory is allocated (and temporarily kept)

[1] For infrequent logging schemes, the only exception is related to the need for keeping data/metadata for at least one logged state image with time t less than GVT, and the events with time in between t and the GVT value, to be able to recover the LP state image to any point in time arbitrarily close, or coinciding with, the GVT value.

to store information related to both speculatively-scheduled events and already committed events.

An additional central point relates to the CPU-scheduling algorithm used to determine which LP, among the ones hosted by a given simulation-kernel instance, must be given control. Several proposals have been made [14–17], but the common choice is the Lowest-Timestamp-First (LTF) algorithm [18], which selects the LP whose next pending event has the minimum timestamp, among all the locally-hosted ones. LTF's advantage is that it does not generate local causality violations. This is because LPs are dispatched similarly to what would happen with a sequential simulator, which imposes a timestamp-ordered sequence of CPU-schedule operations. Hence, rollbacks are generated only due to events scheduled between LPs hosted by different kernel instances.

2 Programming and Object Model in ROOT-Sim

ROOT-Sim [19] is a PDES simulation kernel relying on the optimistic synchronization paradigm. It comes as a static library which can be linked to executables implementing simulation models using the ANSI-C programming standard [20], as if they were completely sequential.

In particular, the user can organize the code in as many functions/files as needed, can perform any I/O operation during the simulation (keeping in mind that I/O operations can degrade performance), can use dynamically-allocated memory to build the simulation state, and so on and so forth. The only exception is that the `volatile` qualifier becomes meaningless (i.e., there is no possibility for a variable to be modified outside the simulation platform). In addition, the model lives in userspace only, so no *NIX system calls should be used. No regular entry point is required for the application-level code (i.e., no `main()` function must be implemented), as entry points for the application code are specified by ad-hoc APIs, which will be discussed in the next section.

The actual simulation is based on events: each LP processes events, and its advancement in the Local Virtual Time (LVT) is connected to their execution, and LPs communicate via messages. ROOT-Sim, as the Time Warp protocol stands, enforces a logical identity between events and messages. This means that a message envelopes an event to be scheduled to some destination LP. Each event (and thus message backing it) is identified by a numerical code, which is defined by the application-level logic. Each type of message is fixed-size. In particular, the application-level code must provide a definition of a `struct` for each event type, where the content of a message (an event) must be specified.

At simulation startup, ROOT-Sim delivers to each LP a special `INIT` event (identified by the reserved code 0) which allows the simulation model to set up its initial configuration. In particular, the model can define its simulation state relying on a sequence of `malloc()` calls, and the initial values can be retrieved by command-line arguments which are delivered as `INIT`-event's payload, resembling the standard ANSI-C `char **argv` vector. During the execution of the `INIT` event, other events can be scheduled at any LP in the system, therefore allowing the actual simulation to start. According to the ROOT-Sim

programming model, the first `malloc()` call issued by each LP during the execution of the `INIT` event is considered as the initial part of the LP's simulation state, and will be later passed via a specific pointer to allow the execution of additional events. This can be overridden by a specific API provided by the platform, as it will be discussed later. Nevertheless, this approach allows LPs' states to arbitrarily grow/shrink during the simulation's exection, just relying on additional `malloc()`/`free()` calls. This unique feature means that the user can produce standard code to design the simulation model, with the only requirement that every additional `malloc`'d memory region must be referred via a pointer in the first `malloc`'d structure for the simulation platform to be able to correctly rollback previous simulation states.

2.1 Supported APIs

The core API to allow communication between application-level code and simulation kernel is very simple. It consists of one call function, `ScheduleNewEvent()`, and two callback functions, `ProcessEvent()` and `OnGVT()`. The callbacks must be necessarily implemented in the simulation model to be compliant with the library. Then, the rest of the code can be implemented in any way, albeit respecting the ANSI-C standard. These functions have the following signature/purpose.

`void ProcessEvent(int me, time_type now, int event_type, void * event_content, void *state)` is the callback that supports the actual processing of simulation events, and it is used by the kernel to give control to the application layer. `me` is the ID of the LP being scheduled, `now` is the current value for the local clock, `event_type` is the numerical code for the event to be processed, `event_content` is the information regarding the event itself, and `state` is the current LP's state. Inside of `ProcessEvent()` the execution is fully speculative, i.e. the events that are executed might be eventually undone. The programmer, nevertheless, is completely unaware of this issue, and can simply implement state transitions within this callback. ROOT-Sim will transparently undo (in case of a detected inconsistency) or commit speculative events (whenever a new GVT value is computed and the commitment horizon is moved forward). The only issue concerning `ProcessEvent()` is the execution of non-rollbackable actions. In fact, if the programmer, e.g., prints some text on the screen during the execution of an event that will be eventually rolled back, the output generated will not be reverted. This is a non-trivial problem associated with speculative execution, even more if transparency is enforced and the programmer is given the freedom to implement its model by relying on standard ANSI-C. ROOT-Sim offers a facility which tries to address this issue (at the cost of some delay in the materialization of the actual output), which will be presented in Section 3.

`void ScheduleNewEvent(int receiver, time_type timestamp, int event_type, void *event_content, int event_size)` is a function that allows injecting a new simulation event within the system, to be destined to whichever simulation object. `receiver` denotes the ID of the destination LP, `timestamp` is the LVT associated with the event to be processed, `event_type`, `event_content`, and `event_size` allow the correct identification and delivery of

the actual event. For efficiency reasons, events are buffered and asynchronously delivered when the execution of the current one is completed. This allows to pack together more events if the destination LP is the same, and prevents delays in the current event's execution. We note that this asynchronous deliver does not affect the correctness of the execution, as ROOT-Sim will order events in the input queue before scheduling the next event to the destination LP. In case the delay created by this internal buffering generates an out-of-order execution at some LP, then the rollback procedure will restore consistency.

`bool OnGVT(void *snapshot, int gid)` is a callback that gives control to the application layer by also providing a committed snapshot of the simulation object. The execution of `OnGVT()` is therefore not speculative, i.e. any action taken within this function will never be undone. This means that, e.g., any I/O operation within this function is perfectly safe, and therefore it can be used to gather statistics on the ongoing simulation. We note that, since the timestamp associated with `snapshot` refers to the committed portion of the computation, it is forbidden to call `ScheduleNewEvent()` within `OnGVT()`, because this might induce a rollback operation of already committed events. In case the user calls `ScheduleNewEvent()` in this callback, a runtime error will be generated. `OnGVT()` additionally implements a distributed termination control: since `snapshot` is a portion S_i of the Committed and Consistent Global State (CCGS) S, according to [21] a global predicate can be locally evaluated on S_i. If the model determines that the simulation is completed for that particular LP, `OnGVT()` can return the `true` value. ROOT-Sim will collect all return values, and in case all the LPs agree, the simulation will stop.

In addition to the core API, ROOT-Sim has a set of additional facilities. First, `void SetState(void *new_state)` allows the LP to manually specify which is its simulation state. It is not mandatory in a simulation-model's implementation, as explained in Section 2, but gives the programmer more freedom.

ROOT-Sim comes bundled with a fully-featured numerical library, which gives the modeler the possibility to generate random numbers drawn from several distributions. So far, the `Random()`, `Expent()` (exponential), `Normal()`, `Gamma()`, `Poisson()`, and `Zipf()` distributions are implemented. This library is crucial in the development of simulation models (and should be used in place of other libraries) because it adheres to the Piece Wise Deterministic paradigm (see [22]), i.e. the same sequence of numbers will be deterministically produced if a rollback operation is performed. The numerical library maintains one seed per each LP, pseudo-randomly drawn by an initial master seed which can be either randomly computed at simulation startup, or manually specified by the user. This gives full control on the model's execution, giving the possibility to re-study the same configuration (determined by the initial master seed) which will give the same final outcome, independently of the actual events' execution pattern.

3 Internal Features

The simulation platform offers several facilities to support model's simulation in the most effective way, and to control the simulation's execution. Configuration

can be specified either at command line, or by relying on an interactive shell which supports execution scripts as well.

The GVT computation interval can be tuned at simulation startup. If the user selects a higher value, the overhead associated with this operation gets reduced, but since the fossil collection operation (see Section 1) is thus executed less often, the overall simulation likely requires more memory to maintain, e.g., older checkpoints. By tweaking this parameter, the user can manually balance the tradeoff between performance and memory consumption.

As mentioned before, at simulation startup ROOT-Sim initializes the internal numerical library starting from a master seed. The user can ask ROOT-Sim to randomly draw this master seed or can specify it manually. In case the seed is randomly chosen, the user can configure ROOT-Sim to store it as the master seed for future runs. In this way, a set of experiments can be conducted on the same events' pattern, therefore comparing the very same model's configuration.

Concerning initialization, ROOT-Sim allows to run the simulation model on a distributed environment, by ultimately relying on MPI for message delivery. Using a description of the machines which can be used to run the simulation (i.e. address for reaching them, and number of CPU-cores), ROOT-Sim transparently sets up the distributed simulation environment. For efficiency reasons, two different mappings between LPs and simulation kernels are available: *block distribution* evenly divides the available LPs across the kernel instances, while *circular* assigns one LP per kernel, in a circular fashion. Depending on the inter-LP interaction, the one that limits at most the number of inter-kernel interactions due to event exchange can be selected, so to reduce the impact on performance.

As for state saving, ROOT-Sim offers a large set of facilities which address both transparency and execution efficiency at the same time. In particular, as mentioned, the simulation state can be scattered across different segments of allocated memory, and log/restore operations can be carried out either in non-incremental [23] or incremental way [24]. Additionally, ROOT-Sim can be configured to switch between these two differentiated modes autonomically and to optimize the checkpointing interval, either by relying on an analytic model [25], or by relying on a genetic algorithm [26].

ROOT-Sim offers the user the possibility to select either the traditional LTF scheduler for event processing selection, or an optimized $O(1)$ variant [27], which selects the next event to be scheduled in a (probabilistic) constant time.

Third-party libraries are almost fully supported. The user is allowed to rely on any third-party library if it is stateless. Support for stateful third-party libraries is currently under development and will be integrated in a future release.

There are several branches of the ROOT-Sim kernel still under development and that will be merged in the main version soon. One relies on multithreading [28] for running in parallel within the same kernel more than one LP and the related housekeeping operations. Another branch supports migration of LPs among different kernel instances depending on the current workload [29], while the orthogonal *load sharing* approach [30] varies the number of CPU-cores assigned to each simulation kernel instance depending on the current workload.

This avoids the costly migration operation, but limits the viability to multi-core machines. Another one allows intersections between LPs' states [31], i.e. global variables can be speculatively accessed and in case of inconsistency they are rolled back as well. Another version supports consistent generation of I/O within the speculative `ProcessEvent()` callback [32], which is done via a proper I/O demon that materializes the output only after discriminating the committed portions and filtering out inconsistent parts of the output stream via optimized stream-management mechanisms.

4 A Code Example

We present here some code snippets implementing a ROOT-Sim application which models a set of N nodes connected as a mesh, sending packets randomly to each other. The first important thing is to define the possible events handled by the model, the content of an event message, and the structure of the state:

```
1  #include <ROOT-Sim.h>
2  #define PACKET 1 // Event definition
3  #define DELAY 120
4  #define PACKETS 1000000 // Termination condition
5
6  typedef struct _event_content_t {
7      time_type sent_at;
8  } event_content_t;
9  typedef struct _lp_state_t{
10     int packet_count;
11 } lp_state_t;
```

In this model we allow just one application-defined event, `PACKET`, which identifies the transit of a packet in the mesh. Then, we must specify the actual events' logic. `ProcessEvent()` is the only entry point for speculative event processing, so we rely on a `switch` construct to demultiplex them:

```
18 void ProcessEvent(unsigned int me, time_type now, unsigned int event, event_t *content,\\
19                   unsigned int size, lp_state_t *ptr) {
20     event_t new_event;
21     time_type timestamp;
22
23     switch(event) {
24
25         case INIT: // must be ALWAYS implemented
26             state = (lp_state_t *)malloc(sizeof(lp_state_t));
27             state->packet_count = 0;
28             timestamp = (time_type)(20 * Random());
29             ScheduleNewEvent(me, timestamp, PACKET, NULL, 0);
30             break;
31
32         case PACKET: {
33             pointer->packet_count++;
34             new_event_content.sent_at = now;
35             int recv = FindReceiver(MESH);
36             timestamp = now + Expent(DELAY);
37             ScheduleNewEvent(recv, timestamp, PACKET, &new_event, sizeof(new_event));
38         }
39     }
40 }
```

```
TOTAL KERNELS.................. : 32
TOTAL PROCESSES............... : 1024
TOTAL EVENTS EXECUTED......... : 169923760
TOTAL COMMITTED EVENTS........ : 96932768
TOTAL ROLLBACKS EXECUTED..... : 2357476
TOTAL ANTIMESSAGES............ : 72208424
AVERAGE ROLLBACK FREQUENCY... : 1.4 %
AVERAGE ROLLBACK LENGTH...... : 30.62 events
EFFICIENCY..................... : 58.11 %
AVERAGE EVENT COST............ : 11.48 us
AVERAGE CHECKPOINT COST...... : 28.327 us
AVERAGE RECOVERY COST........ : 20.110 us
Simulation started at........ : Thu Mar 21 23:23:07 2013
Simulation finished at....... : Thu Mar 21 23:27:49 2013
Computation time............. : 282 seconds
Last GVT value............... : 50828.568372
Average Committed Event-Rate. : 343733.22 events/second
```

Fig. 1. General Statistics Output File

The code logic is fairly simple: upon `INIT` event, the LP's state is `malloc`'d and initialized, and an initial packet is sent to the LP itself. Whenever a `PACKET` event is received, a local counter is increased, and a packet is sent back to a random LP in the simulation environment. Timestamps are computed according to an exponential distribution, exploiting the internal `Expent()` function.

`OnGVT()` is the second callback to be implemented, which performs a local check on the LP's state. If the number of packets passed through the LP is smaller than `PACKETS`, then the simulation cannot be halted yet:

```
50 bool OnGVT(lp_state_t *snapshot, int gid) {
51     if (snapshot->packet_count < PACKETS)
52         return false;
53     return true;
54 }
```

5 Runtime Data

Beyond the statistics that the user can programmatically obtain from the execution of the `OnGVT()` callback, ROOT-Sim has three levels of statistics generation which can be configured at startup. The first (default) level produces a file containing average values for the most common performance metrics on a system-wide scale. An example file is presented in Figure 1. With it, the user can gather information about general execution and overall performance, i.e. the efficiency of the simulation, or the number of events executed.

If the user is interested in the same statistics but at a per-kernel granularity (i.e. not a system-wide average), the second level of statistics generates a text file like the previous one for each kernel instance. This is useful, e.g., to check if the workload of the model is evenly distributed across simulation kernel instances. At the same time, this level produces punctual data during the simulation's

execution in the form ⟨GVT phase number, GVT value, committed events in this phase, cumulated committed events⟩ (one tuple per line). This adds some overhead, which is nevertheless minimal because this operation is performed periodically only during the GVT calculation. This is a very useful information to track performance variations during the execution of the simulation model.

The last (more intrusive) level of statistics generation prints on a per-kernel separate file additional information for each GVT phase, in the form ⟨total events, committed events, rollbacks, average event cost, average checkpoint cost⟩.

References

[1] Fujimoto, R.M.: Parallel discrete event simulation. Communications of the ACM 33(10), 30–53 (1990)
[2] Madhava Rao, D., Thondugulam, N., Radhakrishnan, R., Wilsey, P.: Unsynchronized parallel discrete event simulation. In: Winter Simulation Conference Proc., vol. 2, pp. 1563–1570 (December 1998)
[3] Quaglia, F., Baldoni, R.: Exploiting intra-object dependencies in parallel simulation. Inf. Process. Lett. 70(3), 119–125 (1999)
[4] Fujimoto, R.M.: Exploiting temporal uncertainty in parallel and distributed simulation. In: Proc. of the 13th Workshop on Parallel and Distributed Simulation, pp. 46–53. IEEE Comp. Soc. (May 1999)
[5] Cai, W., Turner, S.J., Lee, B.S., Zhou, J.: An alternative time management mechanism for distributed simulations. ACM Transactions on Modeling and Computer Simulation 15(2), 109–137 (2005)
[6] Jefferson, D.R.: Virtual Time. ACM Transactions on Programming Languages and System 7(3), 404–425 (1985)
[7] Bellenot, S.: Global virtual time algorithms. In: Proc. of the SCS Multiconference on Distributed Simulation, pp. 122–127 (January 1990)
[8] Preiss, B.R., Loucks, W.M., MacIntyre, D.: Effects of the checkpoint interval on time and space in Time Warp. ACM Transactions on Modeling and Computer Simulation 4(3), 223–253 (1994)
[9] Palaniswamy, A.C., Wilsey, P.A.: An analytical comparison of periodic checkpointing and incremental state saving. In: Proc. of the 7th Workshop on Parallel and Distributed Simulation, pp. 127–134. ACM (1993)
[10] Rönngren, R., Liljenstam, M., Ayani, R., Montagnat, J.: Transparent incremental state saving in Time Warp parallel discrete event simulation. In: Proc. of the 10th Workshop on Parallel and Distributed Simulation, pp. 70–77. IEEE Comp. Soc. (May 1996)
[11] Santoro, A., Quaglia, F.: Transparent state management for optimistic synchronization in the High Level Architecture. In: Proc. of the 19th Workshop on Principles of Advanced and Distributed Simulation, pp. 171–180. IEEE Comp. Soc. (June 2005)
[12] West, D., Panesar, K.: Automatic incremental state saving. In: Proc. of the 10th Workshop on Parallel and Distributed Simulation, pp. 78–85. IEEE Comp. Soc. (May 1996)
[13] Carothers, C.D., Perumalla, K.S., Fujimoto, R.M.: Efficient optimistic parallel simulations using reverse computation. ACM Transactions on Modeling and Computer Simulation 9(3), 224–253 (1999)

[14] Som, T.K., Sargent, R.G.: A probabilistic event scheduling policy for optimistic parallel discrete event simulation. In: Proc. of the 12th Workshop on Parallel and Distributed Simulation, pp. 56–63. IEEE Comp. Soc. (May 1998)
[15] Quaglia, F., Cortellessa, V.: On the processor scheduling problem in time warp synchronization. ACM Transactions on Modeling and Computer Simulation 12 (July 2002)
[16] Rönngren, R., Ayani, R.: Service oriented scheduling in Time Warp. In: Proc. of 1994 Winter Simulation Conference, pp. 1340–1346. Society for Computer Simulation (December 1994)
[17] Palaniswamy, A.C., Wilsey, P.A.: Scheduling Time Warp processes using adaptive control techniques. In: Proc. of the 1994 Winter Simulation Conference, pp. 731–738. Society for Computer Simulation (December 1994)
[18] Lin, Y.B., Lazowska, E.D.: Processor scheduling for Time Warp parallel simulation. In: Proc. of the 23rd SCS Multiconference on Advances in Parallel and Distributed Simulation, pp. 11–14. IEEE Comp. Soc. (January 1991)
[19] HPDCS Research Group: ROOT-Sim: The ROme OpTimistic Simulator - v 1.0 (October 2012), `http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/`
[20] Kernighan, B.W., Ritchie, D.M.: The C Programming Language, 2nd edn. Prentice Hall Professional Technical Reference (1988)
[21] Mattern, F.: Efficient algorithms for distributed snapshots and global virtual time approximation. Journal of Parallel Distributed Computing 18(4), 423–434 (1993)
[22] Elnozahy, M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys 34(3), 375–408 (2002)
[23] Toccaceli, R., Quaglia, F.: DyMeLoR: Dynamic Memory Logger and Restorer library for optimistic simulation objects with generic memory layout. In: Proc. of the 22nd Workshop on Principles of Advanced and Distributed Simulation, pp. 163–172. IEEE Comp. Soc. (2008)
[24] Pellegrini, A., Vitali, R., Quaglia, F.: Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In: Proc. of the 23rd Workshop on Principles of Advanced and Distributed Simulation. IEEE Comp. Soc. (2009)
[25] Vitali, R., Pellegrini, A., Quaglia, F.: Autonomic log/restore for advanced optimistic simulation systems. In: Proc. of the Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pp. 319–327. IEEE Comp. Soc. (2010)
[26] Pellegrini, A., Vitali, R., Quaglia, F.: An evolutionary algorithm to optimize log/restore operations within optimistic simulation platforms. In: Proc. of the 4th International ICST Conference on Simulation Tools and Techniques, SIGSIM (2011)
[27] Santoro, T., Quaglia, F.: A low-overhead constant-time LTF scheduler for optimistic simulation systems. In: Proc. of the IEEE Symposium on Computers and Communications, pp. 948–953 (2010)
[28] Vitali, R., Pellegrini, A., Quaglia, F.: Towards symmetric multi-threaded optimistic simulation kernels. In: Proc. of the 26th International Workshop on Principles of Advanced and Distributed Simulation, PADS, pp. 211–220. IEEE Comp. Soc. (August 2012)
[29] Peluso, S., Didona, D., Quaglia, F.: Application transparent migration of simulation objects with generic memory layout. In: Proc. of the 25th Workshop on Principles of Advanced and Distributed Simulation, pp. 169–177. IEEE Comp. Soc. (June 2011)

[30] Vitali, R., Pellegrini, A., Quaglia, F.: Load sharing for optimistic parallel simulations on multi core machines. SIGMETRICS Performance Evaluation Review 40(3), 2–11 (2012)
[31] Pellegrini, A., Vitali, R., Quaglia, F.: Transparent and efficient shared-state management for optimistic simulations on multi-core machines. In: Proc. 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 134–141. IEEE Comp. Soc. (August 2012)
[32] Antonacci, F., Pellegrini, A., Quaglia, F.: Consistent and efficient output-stream management in optimistic simulation platform. In: Proc. of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, pp. 315–326. ACM (May 2013)