A Load-sharing Architecture for High Performance Optimistic Simulations on Multi-core Machines

Roberto Vitali, Alessandro Pellegrini and Francesco Quaglia
DIIAG, Sapienza, Università di Roma

Abstract—**In Parallel Discrete Event Simulation (PDES), the simulation model is partitioned into a set of distinct Logical Processes (LPs) which are allowed to concurrently execute simulation events. In this work we present an innovative approach to load-sharing on multi-core/multiprocessor machines, targeted at the optimistic PDES paradigm, where LPs are speculatively allowed to process simulation events with no preventive verification of causal consistency, and actual consistency violations (if any) are recovered via rollback techniques. In our approach, each simulation kernel instance, in charge of hosting and executing a specific set of LPs, runs a set of worker threads, which can be dynamically activated/deactivated on the basis of a distributed algorithm. The latter relies in turn on an analytical model that provides indications on how to reassign processor/core usage across the kernels in order to handle the simulation workload as efficiently as possible. We also present a real implementation of our load-sharing architecture within the ROme OpTimistic Simulator (ROOT-Sim), namely an open-source C-based simulation platform implemented according to the PDES paradigm and the optimistic synchronization approach. Experimental results for an assessment of the validity of our proposal are presented as well.**

I. Introduction

Parallel Discrete Event Simulation (PDES) techniques are well known for being a classical means to develop simulation systems featuring high performance, which is essential for differentiated contexts, such as symbiotic systems or simulation-based (time-critical) decision making. The basic idea underlying these techniques is to partition the simulation model into several distinct objects, also known as Logical Processes (LPs) [1], which concurrently execute simulation events on clusters, SMP/multi-core machines and/or even desktop grids [2].

In terms of architectural organization, LPs are typically implemented as application-level callbacks (see, e.g., [3]), each of which is in charge of manipulating the LP's state depending on the simulation event dispatched via the callback. Also, multiple LPs are typically run within a same Operating System process, referred to as *simulation kernel*, which is in charge of performing all the housekeeping tasks, such as determining the scheduling sequence for event processing at the various LPs.

The main issue in the design/development of this type of simulation platforms is synchronization, the goal of which is to ensure causally-consistent (e.g. timestamp-ordered) execution of simulation events at each concurrent LP. In literature, several synchronization protocols have been proposed, among which the optimism-oriented ones (e.g. the Time Warp protocol [4]) are highly promising. With these protocols, block-until-safe policies for event processing are avoided, thus allowing speculative computation, and causal consistency is guaranteed through rollback/recovery techniques, which restore the system to a correct state upon the a-posteriori detection of consistency violations.

As hinted, multiple LPs run within a same simulation kernel, and according to its classical organization (see, e.g., [5]), the kernel process is single-threaded. As a consequence, the hosted LPs are dispatched and run on top of an individual CPU-core. By this organization, the typical literature approach aimed at achieving effective parallel/distributed simulation runs, by optimizing the exploitation of the available computing resources, is *load balancing*, which is based on migrating the application load, namely the LPs, amongst different simulation-kernel instances (i.e. different processes).

We hereby propose an orthogonal technique targeted at optimistic PDES systems run on top of multi-core machines, which is based on computing power (expressed in terms of CPU-cores) dynamical reallocation over time towards the different active simulation-kernel instances. Hence, we allow a dynamical scale up/down in the number of worker threads belonging to each kernel instance, which sustain the execution of the whole set of LPs locally hosted by that same instance, depending on whether LPs (dynamically) increase/decrease their computing power demand. Overall, the *load-sharing* approach is pursued, in terms of dynamical redistribution of the whole simulation load across the set of available computing resources. Further, dynamical scale up/down of the amount of worker threads per simulation kernel instance is based on an innovative analytical approach that we also present in this paper.

Our proposal is non-trivial, since (optimistic) simulation platforms typically expose a reduced set of services, internally handled by the simulation-kernel layer via a relatively reduced set of data structures. Hence, the likelihood of conflicts in the access to these data structures upon simultaneous executions in kernel mode by multiple worker threads (each running on top of a different CPU-core) may easily become a bottleneck. To address this issue we borrow from the top/bottom-half programming paradigm, currently employed for handling interrupts within modern, multi-core Operating Systems, to design multi-threaded optimistic simulation kernels guaranteeing minimal length of wait-for-lock phases.

Beyond general design indications, we also present a real implementation of our architectural proposal within ROOT-Sim [6], namely an open-source optimistic simulation platform implemented according to the optimistic synchronization paradigm. Experimental data assessing the viability and the relevance of the provided approach are also presented.

II. Related Work

Given that we aim at optimizing the use of the available computing resources in face of dynamism and fluctuations of the actual workload associated with the LPs involved in the run, our work is naturally related to all the literature solutions that have presented policies for load balancing in the context of either conservative (e.g. [7]) or optimistic simulation (e.g. [8]). Differently from these works (A) we reassign the computing power, instead of the workload, across the active simulation kernel instances, and (B) we rely on an innovative, multi-threading oriented paradigm in order to exploit dynamically scaled up/down power available to each kernel instance. Also, our proposal can be considered as *orthogonal and complementary* to the above results, when considering that we target multi/many-core machines, while the aforementioned load balancing schemes can be used for load redistribution on distributed memory systems.

As for HLA-based simulation platforms (see, e.g., [9]), multi-threading has been used to implement non-blocking interoperability services across federations of simulators. The strongest difference from our approach is that multi-threading has been used to implement sector-specific functionalities, while we use it as a means to overtake differentiated operations (including event processing), depending on dynamical variations of the application level workload. In addition, to the best of our knowledge, changes in the number of worker threads has never been used to perform dynamical optimizations in response to workload's variations. It has only been employed in master-slave simulation architectures to cope with dynamical increase/decrease of the amount of available computing resources (e.g. for simulation platforms running on top of desktop grids [2]). However, in such a context, concurrent threads operate on inherently partitioned data, while we approach multi-threading in the presence of shared (e.g. kernel-level) data structures.

When considering solutions specifically oriented to improve the performance of simulation platforms on multi-core machines, one approach having relations with our proposal can be found in [10]. However, this approach is targeted at a specific architecture, namely the IBM cell processor, while our proposal is general, thus being suited for differentiated multi-core platforms. Also, the work in [10] is oriented to optimize the simulation via task parallelization schemes that are orthogonal to the power-reallocation scheme we present in this article. Similar considerations can be made for other works which address the issue of improving the performance of simulation systems via the exploitation of hardware parallelism offered by GPU architectures (see, e.g., [11]). These approaches are mostly suited for data parallelism while we deal with more general parallelism schemes, which are proper of the PDES paradigm. Also, dynamic power reallocation across different simulation kernel instances is not targeted by those works.

The work in [12] has recently presented an approach for improving optimistic simulations on multi-core machines via the employment of a global schedule mechanism relying on a distributed event queue. Differently from this work, our proposal targets the traditional case of local schedule, characterized by higher scalability thanks to the avoidance of cross-kernel synchronization operations while handling scheduling tasks. Similar considerations can be made when considering simulation architectures like ThreadedWarped [13], which uses a global priority queue.

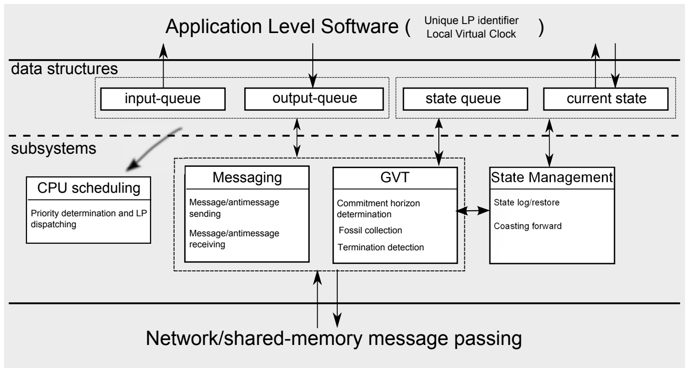


Fig. 1: Reference architecture for optimistic simulation systems.

An approach similar in spirit to the one we propose has been presented in [14]. One main difference between this work and the present proposal lies on that we include in our load-sharing architecture optimized approaches for the handling of statefull CPU schedulers for PDES systems, providing constant-time latency, which are not considered by the work in [14]. Also, we rely on an fully innovative computing power reallocation model, based on estimations of future workload, as opposed to the approach in [14] where CPU-core assignment is determined only on the basis of performance indexes related to past execution.

III. Design Indications

A. Optimistic PDES Systems' Fundamentals

In the optimistic PDES paradigm, LPs are allowed to process their simulation events without preliminary verifying that event-processing does no violate causality rules. Rather, as soon as new events get available for a specific LP, they are eligible for processing. This allows for great exploitation of parallelism, since no block-until-safe rules are adopted. On the other hand, in case a causality violation is revealed, due to the arrival of an event to a destination LP that has already processed other events with larger timestamps (namely events in the future of the arriving one), this LP is rolled back to a correct state value. Further, the effects on other LPs associated with events that were produced by the rolling back LP during the causally incorrect portion of the simulation are also undone by the simulation kernel via proper messaging operations. In Figure 1 we show the reference architectural organization for optimistic simulation systems, as depicted by the seminal paper in [4].

Input and *output message queues* are used to keep track of simulation events exchanged across LPs, or scheduled by an LP for itself. These are typically separated for different LPs, so to afford management costs. For the input queues, these costs are related to both event insertions and event move from the past (already processed) to the future (not yet processed) in case of rollback of a specific LP. The input-queue is sorted by message (event) timestamps, while the output-queue is sorted by virtual send-time, which corresponds to the local virtual clock of the LP upon the corresponding event-schedule operation. As discussed by several works (see, e.g., [15]), the actual implementation of input queues can be differentiated (e.g. heaps vs calendar queues), and possibly tailored to and/or optimized for specific application contexts, characterized by proper event-timestamp patterns (affecting, e.g., the insertion cost depending on the algorithm used to manage the queue). On the other hand, output queues are typically implemented as doubly-linked lists since insertions occur only at the tail (i.e. according to non-decreasing values of the local virtual clock). Also, deletions from the output-queues occur either at the tail or at the head, but not at arbitrary positions. Tail deletions occur upon a rollback operation, which undoes the latest computed portion of the simulation at the involved LP ([1]). Head deletions from the output-queues are related to memory recovery procedures, which we shall detail later on in this section.

A *messaging subsystem* takes care of the exchange of messages/antimessages across different simulation kernel instances, whose content will be then reflected into the input queue of the destination LP.

The *state queue* is the fundamental means for allowing recoverability of the LP's state to a previous snapshot whenever a causal inconsistency is detected (i.e. the LP receives a message with timestamp lower than its current simulation clock, or an antimessage that annihilates an already processed event). The state queue is handled by the *state management subsystem*, the role of which is to save/restore state images ([2]). Additional tasks by this subsystem are related to (i) performing rollback operations (i.e. determining what is the most recent suited state which has to be restored from the log), (ii) performing coasting forward operations (i.e. fictitious reprocessing of intermediate events in between the restored log and the point of the causality violation) and (iii) performing fossil-collection operations (i.e. memory recovery, by getting rid of all the events and state logs which belong to an already-committed portion of the simulation). Several solutions have been presented in literature for optimizing the performance of the state-save/restore subsystem (see, e.g., [17]–[20]) and/or for providing state-log/restore transparency vs the application layer (see, e.g., [21], [22]). The latter aspect is crucial for the usability of the optimistic paradigm, since it relates to relieving the application programmer from the burden of designing/implementing low-level state-management tasks.

The *Global Virtual Time (GVT) subsystem* accesses the message queues and the messaging subsystem in order to periodically perform a global reduction aimed at computing the new value for the commit horizon of the simulation, namely the time barrier currently separating the set of committed events from the ones which can still be subject to a rollback. This barrier corresponds to the minimum timestamp of not-yet-processed or in-transit events. In addition, this subsystem cares about termination detection, by either checking whether the new GVT oversteps a given predetermined value, or by verifying some (global) predicate (evaluated over committed state snapshots), which tells whether the conditions for the termination of the model execution are met. Finally, this subsystem is also in charge of performing the so-called fossil collection procedure, aimed at recovering memory buffers currently keeping obsolete messages and logs, namely those related to the newly-committed portion of the computation. As for GVT computation, termination detection, and fossil collection, the literature offers a plethora of optimized solutions, which are either general-purpose or tailored to specific computing platforms such as shared-memory systems vs clusters (see, e.g., [23], [24]).

Finally, a central point relates to the CPU-scheduling approach used to determine which LP, among the ones hosted by a given simulation-kernel instance, must take control for actual event processing activities. Although several proposals have been made, the common choice is represented by the Lowest-Timestamp-First (LTF) algorithm [25]. It selects the LP whose pending next event has the minimum timestamp, compared to pending next events of the other LPs hosted by the same kernel. Coupled with the traditional single-threaded approach for the implementation of the simulation kernel, LTF has the advantage of avoiding the generation of causality violations across the LPs hosted by the same kernel instance. This is because these LPs are dispatched in a similar way to what would happen on top of a sequential simulation engine, which imposes a timestamp-ordered sequence of CPU-schedule operations for all the events (across all the LPs). Hence, rollbacks can be generated only in relation to events scheduled between LPs hosted by different kernels, which contributes to reduce the amount of rollbacks. Different design/implementation variants for LTF exist, among which a basic (stateless) approach, exhibiting $O(n)$ time complexity, relies on traversing the pending next events across the input queues of all the LPs. Recently, an $O(1)$ statefull approach has been provided [26], which is based on reflecting variations of the priority (i.e. of the next event timestamp) of the LPs into the CPU-scheduler state, which is done in constant-time, and in determining the LP with the highest priority again in constant time by running a query on the current CPU-scheduler state.

B. Our Multi-threaded Proposal

A paradigm shift towards the design/implementation of multi-threaded optimistic simulation kernels, entailing multiple worker threads that can concurrently run any of the LPs hosted on top of the same kernel instance, is non-trivial. In

[1]The output messages at the tail of the output-queue, with send-time greater than the logical time of the causality violation, are marked and sent out towards the original destinations in the form of antimessages, and are then removed from the output-queue. Antimessages are used for annihilating previously sent messages so to notify the receivers about the rollback occurrence at the source LP. Upon their arrival, chained rollback can be generated in case the events carried by the originally sent messages have already been processed by the destination LPs.

[2]Recently, approaches have been provided which substitute or complement log-based state recoverability via reverse computation techniques [16]. In such a case, recoverability also relies on having a reverse version of the application code, which is able to backward apply changes occurred on the LP state since the point of the causality violation.

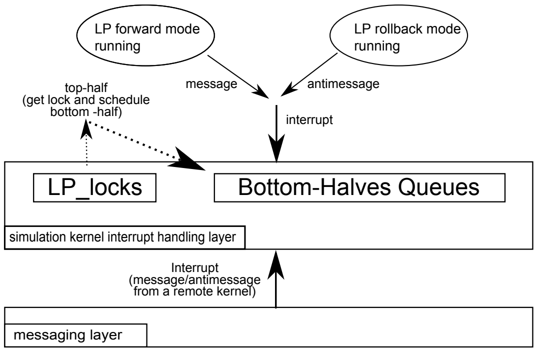
particular, the following two key aspects must be carefully addressed: (A) avoiding synchronization phases while running in kernel mode to become a performance bottleneck, and (B) avoiding loss of locality, which might (unacceptably) degrade the efficiency of the caching hierarchy.

As for point (A), while different worker threads inherently execute according to data partitioning paradigms once entered application mode (since each LP handles its own application-level data structures), care must be taken to avoid "*lock-everything effects*" when running in kernel mode. The risk for these effects is actually due to the relatively reduced set of subsystems forming the optimistic simulation kernel (compared, e.g., to those typically included within the kernel of a general-purpose Operating System), and also to the inherent strict coupling among the LPs (compared, e.g., to the level of coupling of different processes running on top of a conventional Operating System). Most notably, the data structures requiring frequent updates, to be performed coherently via proper kernel-level synchronization mechanisms, are both input and output queues. Essentially, these data structures represent the core of cross-LP dependencies, thus involving update operations caused not only by activities executed by the worker thread currently taking care of running the "queue-owner LP", but also by activities carried out by other worker threads. Synchronizing accesses to these data structures via a conventional locking mechanism would give rise to scalability problems, exactly due to such a strict coupling. Also, it would give rise to critical sections whose duration would depend on the actual time complexity of the queue update operation (which, as mentioned, might even unpredictably depend on the specific event-timestamp pattern).

We note that the access to the LPs' state queues (either for saving or restoring a state image) does not induce thread synchronization issues since the need for state log/recovery operations is only an indirect reflection of cross-LP coupling, caused by events scheduled across the LPs. In other words, a single worker thread operates on the LP state queue at any time, namely the worker thread that has taken care of dispatching the LP for either forward or rollback execution.

As for point (B), we have that a multi-threaded simulation kernel allows virtual addresses related to both application and kernel level data structures (associated with whichever LP) to be, in principle, accessible by any worker thread (since all the worker threads associated with the same simulation kernel instance operate within the same address space). However, such an "unlimited" access policy would cause frequent invalidation/refill of, e.g., the top-level private caches of individual CPU-cores, even when entailing processor affinity schemes involving the worker threads.

1) Addressing Kernel Level Synchronization: The architectural organization we exploit to cope with the reduction of synchronization costs while performing housekeeping operations (see point A) borrows from our proposal in [14]. According to the proposed scheme, any housekeeping task potentially crossing the boundaries of individual LPs' data structures is dispatched according to the same rules employed to structure modern Operating System drivers, by organizing it according to top/bottom-half activities. Specifically, whenever the need for the execution of such a task arises, it takes place as an interrupt to be eventually finalized within a bottom-half module. Hence, upon the interrupt occurrence, we do not immediately finalize the task, thus not immediately locking (or waiting for the lock) on the target data structure. Instead we simply execute a light top-half module which registers the bottom-half function associated with the interrupt finalization within a per-LP bottom-half queue, resembling the Linux task queue. The critical section accessing the bottom-half queue takes constant time since each new bottom half associated with the LP is recorder at the tail of the queue. Also, when the bottom-half tasks currently registered for a given LP are flushed, the corresponding chain of records is initially unlinked from the corresponding bottom-half queue, which is again done in constant time by unlinking the head element within the chain from its base pointer.


LP forward mode running
LP rollback mode running
message
antimessage
top-half (get lock and schedule bottom -half)
interrupt
LP_locks
Bottom-Halves Queues
simulation kernel interrupt handling layer
Interrupt (message/antimessage from a remote kernel)
messaging layer
Fig. 2: Top/bottom halves architecture.

A scheme related to the above architectural organization is provided in Figure 2. Basically, the architecture relies on a spin-lock array, named `LP_locks`, having one entry for each LP hosted by the multi-threaded simulation kernel. `LP_locks[i]` is used to implement the critical section for the access to the bottom-half queue associated with the i-th LP hosted by the kernel, either for inserting a new bottom-half task to be eventually flushed, or for taking care of unlinking the current chain, in order to flush the pending bottom-halves. Techniques for the reduction of the performance impact of spin-lock accesses have also been discussed in [14].

As soon as any worker thread becomes aware of a new message/antimessage destined to the i-th locally hosted LP, it accesses the i-th bottom-half queue within a critical section that performs the insertion of the corresponding message/antimessage delivery task. This occurs when the worker thread either runs some locally hosted LP_j in forward (resp. rollback) mode, and this LP produces a new event (resp. antievent) destined to the locally hosted LP_i. A similar situation occurs when the worker thread performs some receive operation via the messaging layer, which delivers a message/antimessage incoming from some remote kernel instance and destined to a locally hosted LP. As shown in Figure 2, we logically mark all the above circumstances as interrupts, which will be finalized via the bottom-half mechanism.

2) Addressing Locality: In order to cope with locality, we devise the adoption of affinity mechanisms such that a worker thread belonging to a given simulation-kernel instance is not allowed to run every LP hosted by that kernel. Instead, it takes care of running a subset of these LPs, which are currently selected as being affine to the worker thread. In other words, we devise the use of temporary binding mechanisms associating a subset of the locally hosted LPs to a specific worker thread, which is therefore the only thread taking care of running these LPs during a specific wall-clock-time window. Among the activities executed by such a worker thread, we have (i) the flushing of the bottom-half queues associated with its affine LPs, and (ii) the dispatching of these LPs for execution in time interleaved mode. We remark that the binding of a specific LP to a worker thread is not meant to be fixed, but can change over time, also in relation to variations of the amount of worker threads belonging to a given kernel instance. The policy according to which the locally hosted LPs are reassigned to the worker threads which are currently active within the kernel instance will be discussed in Section IV, together with the performance model we use to re-allocate the computing power to the different simulation-kernel instances.

Even though the above approach would seem natural, there is one point that deserves further discussion, and ad-hoc mechanisms targeted at improving the performance of the simulation kernel, in particular on the side of CPU-scheduling operations. As already mentioned, the reference approach for determining the priority of the LPs is the LTF algorithm. In case the adopted LTF scheduler implementation is based on the statefull paradigm presented in [26], which supports constant-time dispatching operation (thus allowing more efficient treatment of simulation models with largely scale-up size) each worker thread should keep its private LTF-scheduler's data structures, and the binding between an LP and a worker thread (say WT_1) would imply that this LP (and its priority) is registered within this private CPU-scheduler state. Now, considering the case in which the LP is dynamically reassigned to a different worker thread (say WT_2), we should atomically remove the LP from the private data structure associated with the LTF scheduler of WT_1, and add it to the data structure instantiating the private LTF scheduler of WT_2. Beyond creating a synchronization point across the different worker threads (with a critical section the length of which depends on the amount of LPs reassigned across the worker threads), this would also adversely affect caching performance since the update operations would anyway invalidate the cached portion of the scheduler state on either WT_1 or WT_2, or would require cache fill with the counterpart LTF-scheduler state, depending on which thread actually carries out the atomic update. This would lead to penalties especially for very large models, where the hierarchical bit-maps organization provided by the statefull approach in [26] would require non-minimal storage.

To overcome these problems, we introduce a mechanism where an additional data structure, namely an array `dispatch_info`, shared across all the worker threads, is used to selectively keep CPU-scheduling information related to each individual LP. Each entry is associated with a distinct

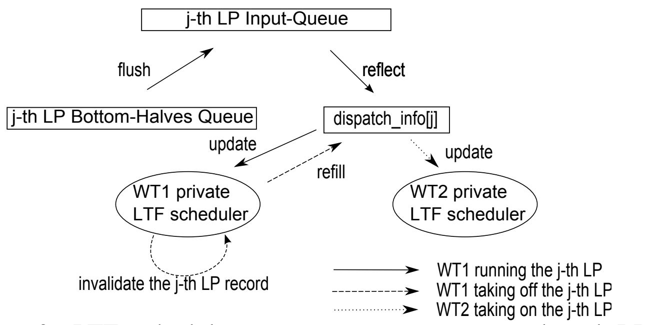

j-th LP Input-Queue
flush
reflect
j-th LP Bottom-Halves Queue
dispatch_info[j]
update
refill
update
WT1 private LTF scheduler
WT2 private LTF scheduler
invalidate the j-th LP record
WT1 running the j-th LP
WT1 taking off the j-th LP
WT2 taking on the j-th LP

Fig. 3: LTF scheduler management upon normal and LP handoff phases.

LP, and implements a very simple state machine keeping track of (a) whether the LP is ready for dispatching (i.e. it has a pending next event) and (b) what is its absolute priority (i.e. the timestamp of such a pending next event). We note that this array can be easily mapped in memory in a way such that each entry results cache-aligned. Also, each worker thread exclusively accesses the entries associated with the LPs that are currently bind to it. Hence, no false cache sharing (with associated cache invalidations) occurs across the worker threads operating within the same kernel instance.

The information kept by `dispatch_info[j]` gets updated and then reflected within the state of the LTF scheduler of the worker thread currently handling the j-th LP (say WT_1) only upon the flush operation of the bottom-half queue associated with this LP (eventually performed by WT_1).

The role of the `dispatch_info[j]` entry becomes central in case the j-th LP gets reassigned to a different worker thread. In particular, upon the reassignment, the source worker thread WT_1 simply re-installs onto `dispatch_info[j]` a state equivalent to the one currently registered within its private LTF scheduler, and then removes the j-th LP from this scheduler. At this point, the destination worker thread (say WT_2) only needs to install the current state kept by `dispatch_info[j]` within its own LTF scheduler state upon taking over the job of running this LP. Overall, no worker thread accesses the LTF scheduler state of any other worker thread, even upon reassigning the LPs across the threads, thus avoiding at all the above discussed cache invalidation problems. An example execution involving the access to `dispatch_info[j]` is shown in Figure 3, where both normal execution phases, with the j-th LP assigned to WT_1, and migration towards WT_2 are shown. As a last observation, the presence of `dispatch_info[j]` as a temporary buffer where CPU-scheduling information are reflected across a sequence of bottom-halves reduces the number of interactions between any worker thread and its private LTF scheduler, thus further increasing locality. In particular, in case the sequence of bottom-halves induces multiple variations of the LP priority, only one of these priorities is eventually reflected within the LTF scheduler.

IV. The Load-Sharing Model

In this section we provide a model for reallocating the available computing resources (i.e. CPU-cores) to the different simulation-kernel instances, depending on variations of the workload associated with the LPs hosted by the kernel. Then we discuss how to bind these LPs to the worker threads operating within that kernel instance.

A. Computing Power Reallocation

Our approach to computing power reallocation relies on a distributed algorithm according to which a given kernel instance activates/deactivates a certain number of worker threads with the aim at maximizing the global event rate (namely, the global amount of committed simulation events per wall-clock-time unit) across all the simulation kernels. This algorithm allocates the C available cores/processors to the K simulation-kernel instances currently running on a machine. We suggest the periodic GVT determination as a good instant to proceed with cores/processors reallocation routines. The power reallocation algorithm works according to the following steps:

Step 1. Each simulation-kernel instance k_i, with $i \in [1, K]$, hosts a set of LPs, the cardinality of which is $numLP^{k_i}$. k_i associates with each LP_l (where $l \in [1, numLP^{k_i}]$) a *workload factor* L_l, which we define as the wall-clock time for advancing the local virtual time of LP_l of one unit. The workload factor L_l is computed by k_i on the basis of the total number of simulation events currently registered as to be processed within the corresponding input-queue, and having timestamp that falls within a distance in the future equal to the last GVT advancement ([3]), normalized to the local virtual time advancement they would produce, weighted by the average CPU time for event processing by LP_l, that is:

$$L_l = \frac{q_l \times \delta_l}{LVT_l^{q_l} - LVT_l^1}$$

where q_l is the amount of pending events within the event-queue of LP_l with timestamps that fall within the interval of interest, LVT_l^i is the timestamp associated with the i-th pending event along the queue, and δ_l is the average CPU requirement for event processing by LP_j along that chain of pending events. We note that, among the above parameters, q_l and LVT_l^i are known in advance, since they are a function of the current state of the input-queue of LP_l. Instead, δ_l is not known in advance since it expresses the average cost for events that have not yet been actually processed. Anyway, it can be approximated by using an exponential mean over already processed events (e.g. events belonging to previous GVT cycles).

Step 2. k_i computes its total workload as:

$$L^{k_i} = \sum_{l=1}^{numLP^{k_i}} L_l$$

[3]Taking into account only those events that are expected to be actually processed according to a future GVT estimation based on the last GVT advancement sample well fits power reallocation performed at each GVT. Additionally, it avoids biases caused by scenarios where computing power is assigned to a simulation kernel that mostly has LPs with events having timestamps much in the future, since these events are not representative of the real simulation workload in the immediate future of the simulation run, and if fast executed they will likely be rolled back.

Step 3. k_i determines the maximum degree of parallelism it can accomplish. This is done in relation to that each LP must execute events serially, i.e., no two worker threads can simultaneously take on the execution of the same LP. Hence, the maximum degree of parallelism is determined by the number of knapsacks of LPs, such that the LPs within a same knapsack would globally induce the same workload as the LP associated with the highest workload factor. Obviously, one knapsack will consist of a unique LP, namely the one associated with the highest workload factor. This task is performed in several steps:

- Workload factors for the LPs hosted by k_i are non-increasingly ordered (let us call them in this order as $L_{l_1}, L_{l_2}, \ldots, L_{l_H}$);
- As hinted, the first (i.e. the highest) factor L_{l_1} is taken as the reference value, and the knapsack formed by LP_{l_1} is defined;
- The other knapsacks are built by aggregating the remaining LPs according to a *0-1 one-dimensional multiple knapsack problem* solving algorithm. This is an NP-Complete problem, whose integral solution is non-trivial. So we decided to follow a procedure such that we allow an ideally infinite set of J sacks to be used, $J = [1, \infty]$, and for each of them, we rely on the *greedy approximation approach* proposed by George Dantzig [27] in which we relax the constraint on the size of the sack, by allowing a maximum "*overflow*" of about 30%. At each step of the algorithm, $\forall i \in [2, H]$ the j-th knapsack's size Kn_j is updated as $Kn_j = Kn_j + L_{l_i}$, and is thus considered full if the size constraint is violated. In that case, a "*new*" sack is created (i.e. j is increased) and begins to be filled, until the total workload is distributed across the whole set of sacks;
- The optimal number of worker threads required by simulation kernel k_i is $W^{k_i} = j$.

Step 4. k_i notifies the tuple $\langle W^{k_i}, L^{k_i} \rangle$ to a *master kernel* ([4]);

Step 5. The master kernel computes the total system's workload:

$$L^{tot} = \sum_{k=1}^{K} L^{k_i}$$

Step 6. A preliminary estimation of the number of cores to be allocated to k_i is performed as:

$$T_{k_i} = \left\lfloor \frac{L^{k_i}}{L^{tot}} \cdot C \right\rfloor$$

enforcing at least $T_{k_i} = 1$.

Step 7. A refined estimation is then calculated:

$$T'_{k_i} = \begin{cases} T_{k_i} & \text{if } T_{k_i} \leq W^{k_i} \\ W^{k_i} & \text{if } T_{k_i} > W^{k_i} \end{cases}$$

Step 8. If $\sum_{k=1}^{K} T'_{k_i} < C$, it means that there are CPU-cores still available. In this case, all the kernels are non-increasingly ordered by resource allocation remainder $R^{k_i} = (W^{k_i} - T'_{k_i})$,

[4]The master kernel can be identified via a distributed consensus protocol, or can be known a-priori (e.g. by specifying it at compile time) depending on the actual simulation platform.

and until there are CPU-cores still available, they are allocated in a round-robin fashion.

Step 9. The master kernel notifies to each kernel k_i the tuple $\langle j, T'_{k_j}\rangle \forall j$.

B. Binding LPs to Worker Threads

At this stage, our load-sharing model relies on a mechanism for binding the $numLP^{k_i}$ LPs hosted by k_i to the T'_{k_i} worker threads currently running within this kernel instance. To fulfil this task, we again exploit knapsack approaches. In particular, the reference knapsack for the binding process is computed as the average load each worker thread is expected to sustain, namely L^{k_i}/T'_{k_i}. Then, every worker thread fills in its own knapsack, by preferentially selecting the LPs that were already affine to that worker thread, which again favors locality. Also in this case we use a tolerance on the knapsack fulfillment according to Dantzig's methodology.

V. Experimental Study

A. Test-bed Platform

We have implemented the proposed load-sharing architecture within ROOT-Sim, which is an open source C/MPI-based simulation package targeted at POSIX systems [6]. 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 (both for processing events and for accessing a committed and globally consistent state image upon GVT calculations), to be implemented according to the ANSI-C standard, and transparently supports all the services required to parallelize the execution.

Among the main features offered by ROOT-Sim we can mention completely transparent recoverability of the state of the LPs achieved through proper hooking of dynamic memory allocation/release [22], plus ad-hoc code instrumentation schemes that allow incremental determination of dirty state portions [28] and that, ultimately, allow dynamical switch between different state log/restore schemes depending on the proper dynamics of the application layer [29].

The single-threaded version of ROOT-Sim also offers innovative transparent supports for LP migration and load balancing [30], which will be considered as a reference for the assessment of the currently presented load-sharing version in terms of the ability to exploit the computing resources offered by a multi-core machine when the actual simulation workload dynamically varies over time.

Integration of the load-sharing approach within ROOT-Sim has been based on `pthread` technology, and on the reorganization of the kernel level data structures in order to (i) provide per-thread private data, and (ii) cache aligned kernel-level memory buffers so to avoid false cache sharing across the worker threads within the same multi-threaded kernel instance. The latter target has been achieved by exploiting the `posix_memalign` API, plus the usage of proper padding schemes within data structures allowing cache alignment for sequences of records, such as arrays of values. As for the accesses to the MPI layer, used to transfer messages across different kernel instances, in our architecture they can be symmetrically issued by any of the worker threads operating within a given kernel instance. Given that the MPI layer does not natively support multi-threading, we have included a wrapper that synchronizes these accesses transparently towards the worker threads via the embedding of critical sections protected by spin-locks.

Finally, the hardware architecture used for testing our proposal is a 64-bit NUMA machine, namely an HP Proliant server, equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. Each processor has 8 CPU-cores (for a total of 32 CPU-cores) that share a 10MB L3 cache (5118KB per each 4-cores set), and each core has a 512KB private L2 cache. The operating system is 64-bit Debian 6, with Linux kernel version 2.6.32.5.

B. Application Benchmarks

In order to evaluate different aspects of the proposed load-sharing architecture, we have conducted experiments on two different application benchmarks, namely *PCS (Personal Communication System)* and *Traffic*. For space constraint we cannot provide a detailed description of these benchmarks, which can anyhow be found in [14].

PCS implements a simulation model of wireless communication systems adhering to GSM technology, where communication channels are modeled in a high fidelity fashion via explicit simulation of power regulation/usage and interference/fading phenomena on the basis of the current state of the corresponding cell. The power regulation model has been implemented according to the results in [31]. This benchmark has been configured in order to provide a constant workload across all the LPs during the whole simulation run. This has been done in order to measure the actual overhead of the load-sharing architecture, while not taking advantages from its ability to reallocate CPU-cores, given the constancy of the workload. For this scenario, we have run experiments with 1024 wireless cells (where one cell is modeled by an individual LP), each one handling 1000 wireless channels.

The Traffic benchmark simulates a complex highway system (at a single car granularity), where the topology is a generic graph, where nodes represent cities or junctions and edges represent the actual highways. Every node is described in terms of car inter-arrival time and car leaving probability, while edges are described in terms of their length. At startup phase, the simulation model is asked to distribute the highway's topology on a given number of LPs. Therefore, every LP handles the simulation of a node or a portion of a segment, the length of which depends on the total highway's length and the number of available LPs.

We have simulated the whole Italian highway network on top of 1024 LPs. We have discarded the highway segments in the islands in order to simulate an undirected connected graph, which allows having the actual workload migrating overall the highway. The topology has been derived from [32], and the traffic parameters have been tuned according to the measurements provided in [33]. This second application benchmark provides a highly dynamic workload that varies

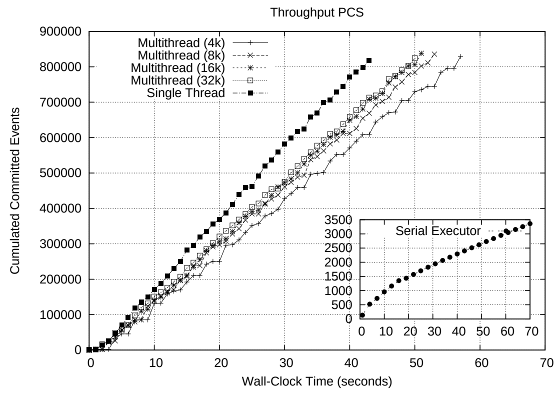


Fig. 4: Simulation throughput for the PCS benchmark.

over time across the involved LPs. This type of benchmark has been used in order to assess the goodness of the load-sharing architecture in terms of its ability to reallocate the computing power depending on the actual needs.

C. Results with the PCS Benchmark

For the PCS benchmark we have measured the cumulated event rate (expressed as the amount of cumulated committed events per wall-clock-time unit), for the case of different configurations of the multi-threaded kernel. In particular, executions with 4, 8, 16, 32 simulation kernels, each one starting with 8, 4, 2, 1 worker thread(s) respectively, have been run. As pointed out before, for PCS the workload is constant and evenly distributed. Hence this benchmark has been adopted to only assess the overhead by the load-sharing architecture (not its ability to cope with dynamic workloads). We have initially set the frequency of call inter-arrival to each cell to the value $\tau_A = 0.8$, which gave rise to average channel utilization factor on the order of 30% and to average event granularity on the order of 30 microseconds. This is relatively fine, thus being a good test case for the evaluation of the overhead by the load-sharing approach.

The results are shown in Figure 4, where all the samples have been obtained as the average over 10 runs all done with different pseudo-random seeds. We have reported different curves, one of which is related to a classical single-threaded execution of the simulation kernel, in which case we always have 32 kernel instances of ROOT-Sim, each one running on top of a different CPU-core. This curve is used as the baseline for the assessment of the overhead by the multi-threaded organization.

The configuration with 4 kernels shows the maximal overhead (expressed via an increase in the wall-clock-time required for committing the same amount of events as for the single-threaded case), which is in the order of 20%. Increasing the number of simulation kernels provides a reduction in the overhead, which looks about 13-15%. We note that, although the workload is constant, small fluctuations due to the probability distribution ruling the generation of the events can arise, which are therefore captured by our architecture. Nevertheless, this sensibility enhances the reassignment's overhead, as long as the time spent in this operation is not rewarded by the new worker threads' configuration. We recall that these data have been achieved for relatively fine event granularity, thus further supporting the viability of our proposal, since applications exhibiting coarser grain events would absorb better the actual overhead by the load-sharing architecture. Also, we note that the parallel approaches provide a super-scalar speedup with respect to a serial executor (based on the calendar-queue [34]), which indicates that they are actually competitive.

In order to assess punctual dynamics associated with the load-sharing approach, we report additional measurements. In particular, to provide quantitative data related to potential variations of the execution locality and its effects, we have decided to focus on three parameters:

- The latency for taking a checkpoint of the LP state.
- The latency for reloading a previously taken checkpoint in case of rollback.
- The event execution latency.

The first two parameters are associated with memory intensive operations, since each log or restore operation entails spanning across the LP state or the log buffer in read mode. They represent therefore a good test case for determining how efficiently these read operations are supported thanks to the effects of the caching hierarchy. On the other hand, the event execution latency is a reflection of the locality expressed by the application, and of how well such a locality is supported via the caching system. These data have been gathered by further expanding the set of values for independent parameters. Specifically, the call inter-arrival time τ_A has been varied between 0.4 and 1.2, thus generating application-level configurations with coarser and finer event granularity, compared to the case $\tau_A = 0.8$. Also, we have performed experiments with multi-threaded kernel configurations entailing up to 32 worker threads.

By the results, shown in Figure 5, we see how both the restore cost and the event processing cost do not show significant variations in any of the considered configurations. At the same time, the log cost provided by the multi-threaded architecture is fairly comparable to the one achieved by the single-threaded architecture for a number of worker threads up to 4 per kernel. On the other hand, an increase of the log latency is noted for the case of 8 or more worker threads. This denotes a slightly reduced locality and/or an increase of thread contention while accessing the malloc library for allocating log buffers, impacting this type of operation. However, this operation is typically executed infrequently (by properly optimizing the tradeoff between checkpointing overhead and restore overhead [29]), thus leading the increased latency to be likely affordable.

On the other hand, we report in Figure 6 the values of the efficiency (namely the percentage of non-rolled back events) for all the investigated configurations. By these data we see how the multi-threaded architecture allows for efficiency values quite close to those achieved with the single-threaded architecture, except for the case where the number of worker threads is set to 16 or 32. By these data we can gain the real picture of why the overhead by the multi-threaded architecture tends to increase when increasing the number of employed worker

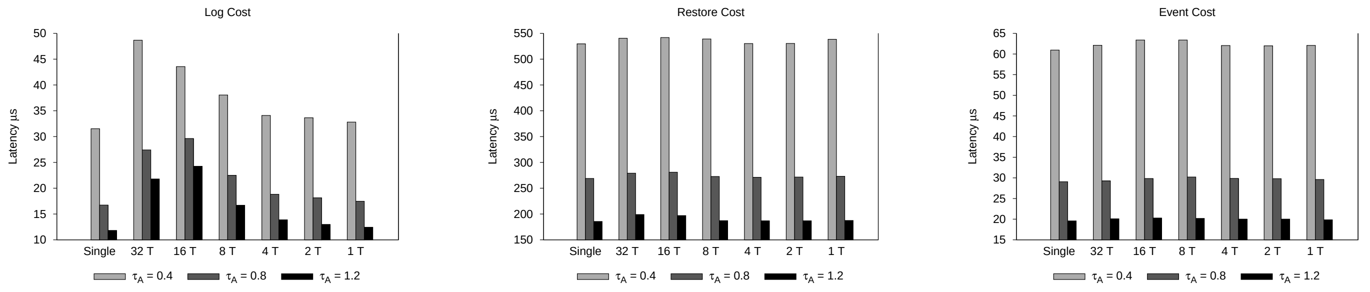


Fig. 5: Costs of log/restore operations and event processing.

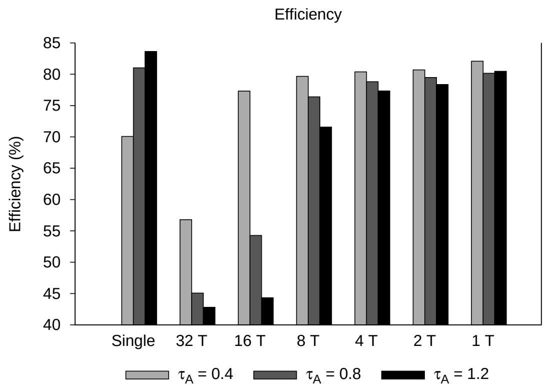


Fig. 6: Efficiency values for the PCS benchmark.

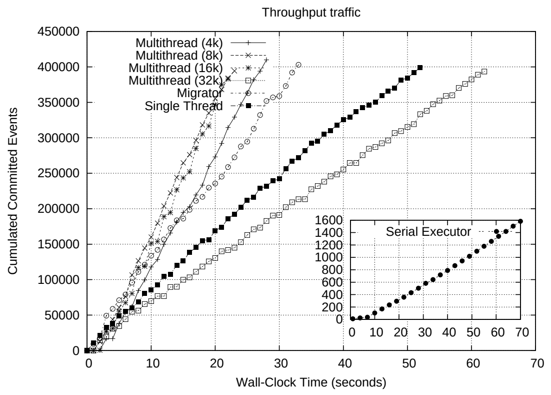


Fig. 7: Simulation throughput for the Traffic benchmark.

threads. Specifically, it is related to the (slightly) increased delay in the delivery of messages that are temporarily buffered into bottom-half queues. This produces an execution scenario where the LPs advance in a less closely related manner in simulation time, which in turn gives rise to an increase of the likelihood of performing useless work to be eventually rolled back due to out of timestamp order processing given the additional delay for incorporating events into the event queues. This phenomenon is clearly more evident for the case of finer grain events (namely τ_A set to 1.2), since more events are allowed to be over-optimistically processed while in transit messages are still buffered within bottom-half queues. We note anyway that this drawback could be addressed by complementary schemes that could flush bottom-half tasks on the basis of, e.g., ad-hoc temporal triggers, so to not induce excessive delay in the final delivery of events/antievents. We plan to investigate along this direction as future work.

D. Results for the Traffic Benchmark

Concerning the results for the Traffic benchmark, which are shown in Figure 7, we have compared the throughput by our multi-threaded architecture with the one by a classical single-threaded organization, with the one related to serial execution of the same application-level software running on top of a calendar-queue scheduler, and with results by a load balancing architecture based on a migration approach, implemented into the same ROOT-Sim platform, as presented in [30].

Again, the parallel approaches provide a super-scalar speedup. The multi-threaded versions of the simulation kernels all provide a speedup wrt the single-threaded one, which ranges in between 40% (for the 4 kernels configuration) and 55% (for the 8 and 16 kernels configuration). In particular, we note that the execution with 4 kernel instances shows a reduced speedup due to several reasons: (i) the re-balancing is more likely to map a worker thread on a core which is not actually sharing any level of cache; (ii) a worker thread can access remote memory with a higher probability (we recall that the experiments have been run on a NUMA machine); (iii) worker threads are more subject to false cache sharing effects.

As for the execution with 32 multi-threaded kernels, the speed down is in the order of 15%. This is related to the fact that in this configuration no actual re-balancing is possible (in fact, each simulation kernel must have at least one worker thread in order to proceed in the simulation run). Therefore, in this configuration we are again measuring the architecture's overhead, which is indeed comparable to the one shown when running the PCS (balanced) benchmark.

The last comparison shown by the plot is the one wrt the load balancing configuration, referred to as migrator. Although we note that this configuration provides a speedup in the order of 40% wrt the single-threaded approach, it's throughput is comparable with the 4 kernels multi-threaded configuration, while the 8 and 16 kernel configurations of the multi-threaded architecture are still 30% faster than the migrator configuration. This is related to the fact that the migrator approach does not assign resources to the simulation kernels, instead it migrates LPs from one instance to the other, entailing complex marshalling and communication operations. We emphasize that these two approaches are orthogonal and do not exclude each other, considering that when relying on clusters, a migration approach merged with a load-sharing approach can result in a significant benefit for the simulation's throughput, as they face different issues.

VI. Future Work and Conclusions

In this work we have presented a new multi-threaded based architectural approach aimed at reassigning the computing power to different optimistic simulation kernels hosted on top of SMP/multi-core machines. Our proposal has been shown to react well on simulations where the workload dynamically changes over time across the involved kernel instances, hence pursuing load-sharing. In addition, it has been shown to add a relatively limited overhead to the simulation environment. Future work entails the coupling of the present work with traditional load balancing approaches, so to allow an optimal usage of resources on top of both shared and distributed memory systems.

References

[1] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990.

[2] A. Park and R. M. Fujimoto, "Optimistic parallel simulation over public resource-computing infrastructures and desktop grids," in *Proceedings of the 12th IEEE International Symposium on Distributed Simulation and Real Time Applications*. IEEE Computer Society, 2008, pp. 149–156.

[3] D. E. Martin, T. J. McBrayer, and 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 - Volume 1: Software Technology and Architecture*. IEEE Computer Society, 1996, p. 383.

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

[5] C. D. Carothers, D. W. Bauer, and S. Pearce, "ROSS: a high performance modular Time Warp system," in *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, May 2000, pp. 53–60.

[6] F. Quaglia, A. Pellegrini, R. Vitali, S. Peluso, D. Didona, G. Castellari, V. Gheri, D. Cucuzzo, S. D'Alessio, and T. Santoro, "ROOT-Sim: The ROme OpTimistic Simulator," Oct. 2011. [Online]. Available: http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/

[7] A. Boukerche and S. K. Das, "Dynamic load balancing strategies for conservative parallel simulations," in *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 1997, pp. 20–28.

[8] D. W. Glazer and C. Tropper, "On process migration and load balancing in Time Warp," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 3, pp. 318–327, 1993.

[9] L. Mellon and D. West, "Architectural optimizations to advanced distributed simulation," in *Proceedings of the 27th Winter Simulation Conference*, 1995, pp. 634–641.

[10] Q. Liu and G. Wainer, "Multicore acceleration of discrete event system specification systems," *SIMULATION*, vol. 88, no. 7, pp. 801–831, july 2012.

[11] T. Hamada and K. Nitadori, "190 TFlops astrophysical N-body simulation on a cluster of GPUs," in *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–9.

[12] L. Chen, Y. Lu, Y. Yao, S. Peng, and L. Wu, "A well-balanced Time Warp system on multi-core environments," in *Proceedings of the 25th Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 2011, pp. 1–9.

[13] R. J. Miller, "Optimistic parallel discrete event simulation on a beowulf cluster of multi-core machines," Master's thesis, University of Cincinnati, 2010.

[14] R. Vitali, A. Pellegrini, and F. Quaglia, "Towards symmetric multi-threaded optimistic simulation kernels," in *Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 6 2012, pp. 211–220.

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

[16] C. D. Carothers, K. S. Perumalla, and R. Fujimoto, "Efficient optimistic parallel simulations using reverse computation," *ACM Transactions on Modeling and Computer Simulation*, vol. 9, no. 3, pp. 224–253, july 1999.

[17] B. R. Preiss, W. M. Loucks, and D. MacIntyre, "Effects of the checkpoint interval on time and space in Time Warp," *ACM Transactions on Modeling and Computer Simulation*, vol. 4, no. 3, pp. 223–253, Jul. 1994.

[18] F. Quaglia, "A cost model for selecting checkpoint positions in Time Warp parallel simulation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 4, pp. 346–362, Feb. 2001.

[19] F. Quaglia and A. Santoro, "Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 6, pp. 593–610, june 2003.

[20] R. Ronngren and R. Ayani, "Adaptive checkpointing in Time Warp," in *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*. Society for Computer Simulation, Jul. 1994, pp. 110–117.

[21] R. Ronngren, M. Liljenstam, R. Ayani, and J. Montagnat, "Transparent incremental state saving in Time Warp parallel discrete event simulation," in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, May 1996, pp. 70–77.

[22] R. Toccaceli and F. Quaglia, "DyMeLoR: Dynamic Memory Logger and Restorer library for optimistic simulation objects with generic memory layout," in *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 2008, pp. 163–172.

[23] D. Bauer, G. Yaun, C. D. Carothers, M. Yuksel, and S. Kalyanaraman, "Seven-o'clock: A new distributed GVT algorithm using network atomic operations," in *Proceedings of the 19th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 2005, pp. 39–48.

[24] R. M. Fujimoto and M. Hybinette, "Computing global virtual time in shared-memory multiprocessors," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 4, pp. 425–446, Oct. 1997.

[25] Y. B. Lin and E. Lazowska, "Processor scheduling for Time Warp parallel simulation," in *Advances in Parallel and Distributed Simulation*, 1991, pp. 11–14.

[26] T. Santoro and F. Quaglia, "A low-overhead constant-time LTF scheduler for optimistic simulation systems," in *Proceedings of the IEEE Symposium on Computers and Communications*, 2010, pp. 948–953.

[27] G. B. Dantzig, "Discrete-variable extremum problems," *Operational Research*, no. 5, 1957.

[28] A. Pellegrini, R. Vitali, and F. Quaglia, "Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects," in *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 2009, pp. 45–53.

[29] R. Vitali, A. Pellegrini, and F. Quaglia, "Autonomic log/restore for advanced optimistic simulation systems," in *Proceedings of the 18th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE Computer Society, 2010, pp. 319–327.

[30] S. Peluso, D. Didona, and 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, june 2011, pp. 169–177.

[31] S. Kandukuri and S. Boyd, "Optimal power control in interference-limited fading wireless channels with outage-probability specifications," *IEEE Transactions on Wireless Communications*, vol. 1, no. 1, pp. 46–55, 2002.

[32] "Atlante stradale italia," http://www.automap.it/.

[33] http://www.autostrade.it/studi/studi_traffico.html.

[34] R. Brown, "Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem," *Communications of the ACM*, vol. 31, pp. 1220–1227, October 1988.