Load Sharing for Optimistic Parallel Simulations on Multi-core Machines

Roberto Vitali
DIAG – Sapienza, University of Rome

Alessandro Pellegrini
DIAG – Sapienza, University of Rome

Francesco Quaglia
DIAG – Sapienza, University of Rome

ABSTRACT

Parallel Discrete Event Simulation (PDES) is based on the partitioning of the simulation model into distinct Logical Processes (LPs), each one modeling a portion of the entire system, which are allowed to execute simulation events concurrently. This allows exploiting parallel computing architectures to speedup model execution, and to make very large models tractable. In this article we cope with the optimistic approach to PDES, where LPs are allowed to concurrently process their events in a speculative fashion, and rollback/recovery techniques are used to guarantee state consistency in case of causality violations along the speculative execution path. Particularly, we present an innovative load sharing approach targeted at optimizing resource usage for fruitful simulation work when running an optimistic PDES environment on top of multi-processor/multi-core machines. Beyond providing the load sharing model, we also define a load sharing oriented architectural scheme, based on a symmetric multi-threaded organization of the simulation platform. Finally, we present a real implementation of the load sharing architecture within the open source ROme OpTimistic Simulator (ROOT-Sim) package. Experimental data for an assessment of both viability and effectiveness of our proposal are presented as well.

1. INTRODUCTION

Parallel Discrete Event Simulation (PDES) is well known for being a classical means to develop simulation systems featuring high performance. This can be relevant for differentiated contexts, such as when virtual and real worlds interact for either training purposes (see, e.g., [24]), or for system prediction/audit in scenarios where components are still under design/development, thus externalizing their behavior via simulation of corresponding models (see, e.g., [3]).

The idea underlying PDES techniques is to partition the simulation model into several distinct objects, also known as Logical Processes (LPs) [9], which concurrently execute simulation events, thus allowing for exploitation of parallelism in the underlying hardware architecture. The main problem 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 [13]) are highly promising. With these protocols, block-until-safe policies for event processing at the LPs are avoided, thus allowing speculative computation, which is reflected into great exploitation of parallelism. At the same time, causal consistency is guaranteed through rollback/recovery techniques, which restore the system to a correct state upon the a-posteriori detection of consistency violations. These are originated when LP_a schedules a new event destined to LP_b having a timestamp lower than the one of some event already processed by LP_b.

The traditional approach to the design and actual implementation of optimistic simulation platforms consists in having multiple LPs being run within a same single-threaded simulation-kernel process (see, e.g., [6]). As a consequence, the LPs hosted by this process are dispatched and run on top of an individual CPU-core, according to a classical time-interleaved mode. By this organization, the typical literature approach aimed at achieving effective simulation runs, by optimizing the exploitation of the available computing resources, is *load balancing*. This technique is based on migrating the application load (i.e. LPs) amongst different simulation-kernel processes while the run is in progress. No other means to dynamically re-balance the load can be employed since each simulation-kernel process has fixed computing power allocated to it, namely one CPU-core.

Clearly, this approach needs to rely on a distributed protocol in order to determine whether a re-balance action is required. This typically maps onto a master/slave protocol, with $O(k)$ message complexity, where the master simulation-kernel process gathers statistics on the current load profile from the other $k-1$ processes, and then notifies the new configuration to be adopted, if any. Computing the current load profile typically requires to sort the LPs according to some reward metric (e.g. the percentage of non-rolled back work) so to be able to determine which LPs need to be migrated. This can be achieved with $O(n \cdot log\ n)$ complexity, where n expresses the number of LPs.

However, beyond the above costs, actual re-balance additionally requires reinstalling onto the destination process' address space the image of any migrated LP. This operation has per-LP latency Δ_m whose lower bound is:

$$\Omega(\Delta_m) = \delta_t \cdot \left[S_{state} + \sum_{i=1}^{N_P} S_{evt}^i \right] \quad (1)$$

where we denote with: δ_t the average per-byte transfer time between source and destination simulation-kernel processes; S_{state} the migrating LP's state size; N_P the number of pend-

ing events for the migrating LP; S^i_{evt} the size of the i-th pending event for the migrating LP.

With the above lower bound, we do not intend to capture aspects associated with, e.g., event-queue implementation and related scan/update costs. We do not even include the latency for transferring data needed to support correct recovery in case of rollback[1]. Anyway, by Equation (1), there is a clear dependency between the actual cost for supporting re-balance and the complexity of the simulation model, in terms of both size of the state of individual LPs to be migrated and event density along the simulation time axis.

We hereby propose an orthogonal approach targeted at optimistic PDES systems run on top of multi-core machines, which is based on computing power (expressed in terms of CPU-cores) dynamic reallocation over time towards the different active simulation-kernel processes. This is achieved by scaling up/down the number of worker threads operating within each kernel instance, depending on whether locally hosted LPs increase/decrease their computing power demand. Overall, we put in place a *load sharing* approach that ultimately redistributes the whole simulation load across the whole set of available computing resources, without the need for actual migration of the LPs across the different kernel instances. The redistribution rule is based in our proposal on an innovative algorithm/model specifically targeted at capturing resource productive usage in the context of optimistically synchronized simulators.

Although this approach requires a distributed protocol similar in complexity to the aforementioned master/slave one, plus some local sorting of LPs' related information, for determining whether and how to reconfigure the system, it does not pay any LP transfer cost upon system's reconfiguration. In fact, the only additional paid costs relate to worker-thread suspension/reactivation (and associated cache refill), which are anyhow not directly dependent on the aforementioned complexity of the simulation model (e.g. in terms of event density along the simulation time axis).

Obviously, the final effectiveness of such an approach depends on how well the worker threads can concurrently operate within a same simulation-kernel process during normal execution phases. To this end, we also provide a reference architectural organization, based on a symmetric multi-threading paradigm, which makes inter-thread synchronization costs affordable. Further, we present and evaluate a real implementation of our proposal within the open source ROme OpTimistic Simulator (ROOT-Sim) package. By the evaluation we demonstrate both the viability of the symmetric multi-threading paradigm, when actuated according to our architectural indications, and the effectiveness of load sharing, when supported via the proposed methodology.

The remainder of this article is structured as follows. In Section 2 we provide the reader with an overview of core aspects related to optimistic simulation. Work related to our proposal is discussed in Section 3. The load sharing methodology is presented in Section 4. The architectural organization based on symmetric multi-threading is depicted in Section 5. Experimental results are reported in Section 6.

[1]This data includes, e.g., already processed but uncommitted events (which might be required to be reprocessed in case of rollback of the LP after the migration phase) and state log information to correctly reconstruct past LP's state snapshots onto the destination kernel-process.

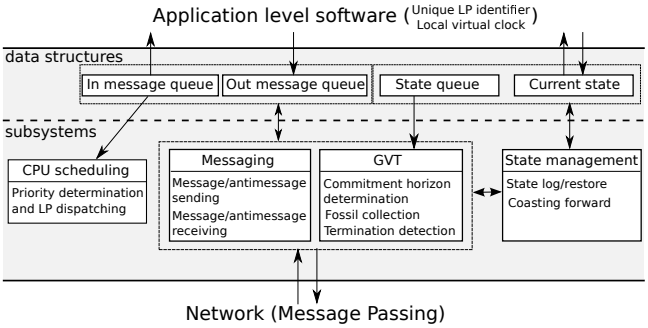

Figure 1: Reference architectural organization.

2. OPTIMISTIC SIMULATION OVERVIEW

The seminal paper in [13] provides basic principles underlying the optimistic paradigm, including a reference architectural organization, which we schematize in Figure 1. *Input and output message queues* are used to keep track of simulation events exchanged across LPs, or scheduled by an LP for itself. They are typically separated for different LPs, so to afford management costs. For the input queues, these costs are related to both event insertions and, e.g., event move from the past (already processed) to the future (not yet processed) in case of rollback. 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. The actual implementation of input queues can be differentiated (e.g. heaps vs calendar queues [5]), and possibly tailored to and/or optimized for specific application contexts, characterized by proper event-timestamp patterns (affecting the insertion cost depending on the algorithm used to manage the queue). On the other hand, insertions in the output queues occur only at the tail, hence these queues are typically implemented as doubly-linked lists, providing $O(1)$ complexity for insertion operations. Also, deletions from output queues occur either at the tail or at the head (again in $O(1)$ time for the case of double-linked lists). The former occur upon a rollback operation. In particular, the output messages at the tail of the output-queue with send-time greater than the logical time associated with the causality violation are marked, sent out towards the original destination in the form of anti-messages — used to annihilate previously sent messages and inform the original receiver of the occurred rollback[2] — and then removed from the output-queue. The latter are related to memory recovery procedures, which we shall detail later on.

A *messaging subsystem* receives incoming messages from other simulation kernel instances, the content of which will be then reflected within the input queue of the destination LP, locally hosted by the recipient kernel instance. Also, this subsystem is used to notify output messages (i.e. newly scheduled events) to LPs hosted by other kernel instances, or the aforementioned anti-messages.

The *state queue* is the fundamental means for allowing a correct restore of the LP state to a previous snapshot whenever a causality inconsistency is detected. The state queue is handled by the *state management subsystem*, the role of which is to save/restore state images. Additional tasks by

[2]Chained rollback can arise if the events to be annihilated have already been processed by the destination LPs.

this subsystem are related to (i) performing rollback operations (i.e. determining what is the most recent 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).

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 rollback. This barrier corresponds to the minimum timestamp of not yet processed or in-transit events/antievents. In addition, this subsystem cares about termination detection, typically by checking whether the new GVT oversteps a given value, and is also in charge of triggering the fossil collection procedure.

Finally, a CPU-scheduling approach is used to determine which among the LPs hosted by a given simulation-kernel instance must take control for actual event processing activities. Among several proposals [22], the common choice is represented by the Lowest-Timestamp-First (LTF) algorithm [15], which 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.

3. RELATED WORK

Given that we aim at optimizing the use of computing resources in face of dynamism and fluctuations of the workload associated with the LPs, our work is naturally related to literature solutions that have presented policies for load balancing in the context of either conservative (e.g. [4]) or optimistic simulation (e.g. [10]). As already hinted, the main difference between our proposal and these works is that we reassign computing power, instead of workload, across the active simulation-kernel instances. Hence, 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 (e.g. clusters).

Since we rely on an innovative architectural organization based on symmetric multi-threading, our proposals has also relations with proposals explicitly targeted at improving performance of simulation systems by relying on concurrent threads operating within a same process. As for HLA-based simulation platforms (see, e.g., [17]), multi-threading has been used to implement non-blocking interoperability services across federations of simulators. The clear 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 dynamic variations of the application level workload. In addition, to the best of our knowledge, changes in the number of worker threads within a process has never been used to perform dynamic optimization in response to workload's variations. It has only been employed in master-slave simulation architectures to cope with dynamic increase/decrease of the amount of available computing resources (e.g. for simulation platforms running on top of desktop grids [19]). However, in such a context, concurrent threads operate on inherently partitioned data, while we approach multi-threading in the presence of shared data structures within the simulation-kernel process.

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 [16]. 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 [16] 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 [7] 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. This proposal has been evaluated with a multi-core machine comprising 8 cores. 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, as we also show via experimentation on a multi-core server equipped with a total of 32 cores. Similar considerations can be made when considering simulation architectures like ThreadedWarped [18], which uses a global priority queue.

4. THE LOAD SHARING APPROACH

Let us denote with C_{tot} the amount of available CPU-cores, and let us assume that we have a set of K_{tot} active simulation-kernel instances in the simulation run, with $K_{tot} < C_{tot}$. Our first objective is to determine the amount of CPU-cores C_i to be assigned to each kernel instance k_i (with $i \in [1, K_{tot}]$) for a given wall-clock-time window, so to improve resource exploitation for fruitful processing.

In our proposal, the re-evaluation of C_i values is carried out periodically, upon computing a new GVT value (or after a set of subsequent GVT computations) since it exploits information on the event rate (committed events per wall-clock-time unit) achieved by each kernel instance k_i, which we denote as evr_i. This quantity is a measure for the fruitful (non-rolled back) amount of simulation work carried out by each kernel instance. In an ideal scenario where the efficiency is maximized (i.e. where the undone computation is negligible), each kernel instance k_i should use an amount of computing power that suffices to execute exactly evr_i events per wall-clock-time unit. In fact, an excess of computing power could lead to over-optimism and hence to rolled back computation, thus moving run-time dynamics far from the above depicted ideal case. So the idea behind the determination of C_i values is to dynamically assign an amount of CPU-cores to kernel k_i which is proportional to the actual requirements of k_i for the achievement of its relative event

rate, compared to the one by the other kernel instances. To also take into account possible differences in the event granularity across the LPs hosted by different kernel instances, which is the indicator of the real usage of computing power for committing the events, the evr_i metric can be refined by weighting it via the average CPU time required for processing the committed events on a specific kernel instance k_i, which we denote as Δ_i. Hence we express the weighted event rate as $wevr_i = evr_i \times \Delta_i$.

In other words, $wevr_i$ values observed during the last wall-clock-time period express the relative CPU requirements of each kernel instance in order to carry out productive simulation work, in relation to the activities of the other kernels and to actual synchronization dynamics. Hence, assigning a computing power proportional to the relative weighted event rate would tend to lead to the situation where each kernel instance allows advancing its LPs in simulation time in a "synchronization suited" manner according to what the other kernels are able to do on their own. This part of the dynamic reallocation scheme would therefore tend to avoid significant presence of overoptimistic kernel instances during the various phases of the run.

It is anyway typical that performance can be further enhanced even in cases where the efficiency is already maximized (or optimized), for example by further reassigning the computing power depending on the real weight of the workload associated with the hosted LPs. As an example, for loosely synchronized models, we may have two or more groups of LPs that do not interact, or stop interacting during the run (hence eventually not directly impacting synchronization and efficiency), exhibiting different speed of advancement in simulation time due to, e.g., different weights of the corresponding events in terms of CPU requirements. In such a case, the completion of the simulation would be delayed by the slowest group. Therefore, within the dynamic scheme for resource assignment, an increase of computing power should also be envisaged for all those kernel instances exhibiting larger CPU requirements to advance in simulation time. To this end we include in our scheme the parameter $wcta_i$, which indicates the wall-clock-time required by kernel k_i to advance a single simulation time unit.

Finally, the amount of cores C_i to be assigned to kernel k_i should anyway be bounded by the maximum degree of parallelism that can be accomplished by k_i, which is a function of the amount of locally hosted LPs. In fact, each LP is an intrinsically sequential entity, which is not further parallelized, thus not being allowed to simultaneously use multiple CPU-cores for its execution.

Overall, we devise the following rules for dynamically defining the amount of CPU-cores to be reassigned to each kernel instance k_i in order to optimize the usage of the available computing power:

1. For each simulation-kernel instance k_i we compute the parameter $\alpha_i = \frac{wevr_i}{\sum_{j \in [1, K_{tot}]} wevr_j}$.
2. A first estimation of C_i is then evaluated as $\widehat{C}_i = max(\lfloor \alpha_i \cdot C_{tot} \rfloor, 1)$.
3. For each kernel instance k_i for which the condition $\widehat{C}_i \geq N_i$ is verified (where N_i identifies the number of LPs hosted by k_i), then C_i is definitively set to N_i. In fact, additional CPU-cores could not be effectively exploited for parallelization of the locally hosted LPs.
4. At this point, there could be some CPU-cores left to be assigned, which we decide to assign on the basis of (A) the request for allocation remainder of kernel k_i, namely $r_i = max([(\alpha_i \cdot C_{tot}) - \widehat{C}_i], 0)$ and (B) the parameter $wcta_i$. In particular, we order the kernels for which the finalization of C_i values still needs to be performed (so the ones already finalized in point 3 are excluded) according to decreasing values of the product $r_i \cdot wcta_i$, and we assign the remaining CPU-cores according to a round-robin rule following the priority defined by such an ordering.

Each of the above steps is an implementation of the rationales discussed above in terms of suited CPU-core assignment vs specific performance aspects. However, once selected final C_i values, we need to determine how to optimize the usage of the assigned CPU-cores within each single simulation-kernel instance k_i. This problem translates into defining which LPs locally hosted by k_i needs to be assigned to each of the C_i worker threads running in parallel within kernel instance k_i. We will refer to the assigned LPs as being *affine* to the worker thread, and still scheduled for event execution along this thread according to LTF.

For the j-th LP hosted by kernel k_i, which we denote as $LP^j_{k_i}$, we compute the total amount of CPU-time required for committing its events during the last observation period. We refer to this metric as $cpu^j_{k^i}$. The maximum $cpu^j_{k_i}$ value across all the locally hosted LPs represents in our scheme a reference knapsack, and the corresponding $LP^j_{k_i}$ is assigned to a given worker thread. Then we exploit the greedy approximation approach proposed by George Dantzig in [8] which allows a maximum "*overflow*" of about 30% over the reference knapsack, in order to build the other knapsacks of LPs (hence knapsacks characterized by sums of $cpu^*_{k_i}$ values) to be assigned to the remaining worker threads. We can do this by applying a variant of the original scheme, where the knapsacks are filled according to a round-robin approach. The procedure is then iterated until no more LP needs to be further bind to any worker thread within k_i.

As a preliminary note to the complexity analysis presented in the subsequent section, the computation of $wevr_i$ and $cpu^j_{k_i}$ values can be embedded within the fossil collection algorithm. In particular, while scanning the input queues of the LPs for releasing the buffers related to the already committed portion of the simulation, per-event execution costs (typical logged while processing the events within the same buffers as a form of audit) can be accessed and accumulated to determine the actual values of the parameters $wevr_i$ and $cpu^j_{k_i}$. Hence, these values can be made available to the algorithm supporting the above described load sharing policy with no variation of the asymptotic cost of fossil collection.

4.1 Asymptotic Costs Analysis

Solving the load sharing model can ultimately rely on a distributed master/slave protocol were every kernel instance k_i sends to the master kernel a message containing the values of the parameters $wevr_i$ and $wcta_i$, and the master kernel sends to k_i a message notifying the newly computed value of C_i. This entails $O(K_{tot})$ message complexity, namely linear complexity vs the number of kernel instances.

The local execution cost at the master kernel for the determination of C_i values, associated with steps 1–4 described above, relates to performing per-kernel analysis of the statistics collected during the master/slave communication phase, and to sorting the kernel instances on the basis of $r_i \cdot wcta_i$ values (see step 4). This leads to an asymptotic $O(K_{tot} \cdot$

$log\ K_{tot}$) complexity.

Once received the notification of the computed C_i value from the master, every kernel instance k_i must sort the locally hosted LPs on the basis of $cpu^j_{k_i}$ values, which entails $O(N_i \cdot log\ N_i)$ time, and must then solve a 0-1 Knapsack's problem, which can be done in pseudo-polynomial time in the number N_i of locally hosted LPs. Hence we get an overall cost of $O(N_i \cdot log\ N_i)$ for local operations to be executed on each simulation kernel instance k_i.

Given that $N_i \geq 1$, we get $\sum_{i \in [1,K_{tot}]} N_i \geq K_{tot}$, hence $O(K_{tot} \cdot log\ K_{tot})$ is bounded by $O(N_{tot} \cdot log\ N_{tot})$, where N_{tot} represents the sum of individual N_i values, namely the total amount of LPs within the run.

In the end, we get that the determination of the new configuration can be achieved with linear message complexity, vs the number of kernel instances, and $O(n \cdot log\ n)$ local processing complexity vs the number n of LPs within the simulation model. This is the same complexity as for typical load balancing schemes, whose asymptotic costs have been discussed in Section 1. However, as already pointed out in the same section, our load sharing approach does not entail actual LP transfer operations, but only activation/deactivation of worker threads within the different kernel instances. We remark again that the cost of this operation does not directly depend on the complexity of the simulation model, in terms of state size of the LPs and event density in simulation time, as instead it occurs for LP transfer operations proper of load balancing approaches.

5. ARCHITECTURAL ASPECTS

The core requirement for the effectiveness of the load sharing approach, is related to how efficiently the different worker threads running within the same simulation-kernel process (hence within the same address space) can synchronize with each other. Specifically, while different worker threads inherently execute according to data partitioning paradigms once entered application mode (since, in accordance with what specified in [13], each LP handles its own application-level data structures), care must be taken to avoid excessive synchronization costs when running housekeeping tasks involving shared data structures.

Most notably, the shared data structures requiring frequent updates, to be performed coherently via proper synchronization mechanisms, are the input queues of the LPs. Essentially, these data structures represent the core of cross-LP dependencies, thus involving update operations caused not only by the activities executed by the worker thread currently taking care of running the "queue-owner LP", but also by the activities carried out by worker threads taking care of running other LPs. Synchronizing the access to these data structures via a conventional locking mechanism would give rise to scalability problems, exactly due to such a strict coupling. Further, it would give rise to critical sections whose duration would depend on the actual time-complexity of the queue-update operation[3].

[3]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/restore operations is only an indirect reflection of cross-LP coupling, caused by events scheduled across the LPs. In other words, a single worker thread is allowed to safely operate on the state queues of its affine LPs at any time, since it is the only worker thread that can take care of dispatching those LP for either forward or rollback execution. Similar considerations can be made for the output queues, which are essentially used for auditing the messages sent out by the LPs in order to undo them via antimessages in case of rollback. On the other hand, the rollback operation and the generation of antimessages, via consultation of audit information within the output queue, are performed (if requested) by the unique worker thread for which a specific LP is currently affine.

The architectural organization we propose in this paper to cope with the reduction of synchronization costs borrows from the design principles proper of multi-processor/multi-core Operating Systems. Specifically, all the worker threads that are active within the same kernel instance operate symmetrically, by having access to any housekeeping functionality. On the other hand, 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. Hence, whenever the need for the execution of such a task arises, it (logically) takes place as an interrupt to be eventually finalized within a bottom-half module. More in details, 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 (and its parameters) 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 recorded 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[4]. Given that the access to the LP bottom-half queue represents in our architectural organization the only frequently occurring synchronization point, constant-time for the corresponding critical sections directly leads to minimizing synchronization costs.

The schematization of our proposal is presented in Figure 2. Basically, our approach can be supported by relying on a spin-lock array, named `LP_LOCKS`, having one entry for each LP hosted by the multi-threaded simulation-kernel. `LP_LOCKS[j]` is used to implement the critical section for the access to the bottom-half queue associated with the j-th LP hosted by any kernel k_i, namely $LP^j_{k_i}$, 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.

Let us now depict when (logical) interrupts to be handled via this type of organization occur. Basically, an interrupt occurs as soon as any worker thread currently active within kernel k_i becomes aware of a new message/antimessage destined to some locally hosted $LP^j_{k_i}$. In such a case, the worker thread needs to accesses the j-th bottom-half queue within a critical section that performs the insertion of the corresponding message/antimessage delivery task. To provide additional details, awareness by a worker thread of a new message/antimessage destined to a locally hosted LP arises

[4]Actual data structure updates can be safely performed out of the critical section, provided that a single worker-thread at any time is in charge of flushing the bottom-halves of any LP that is affine to it.

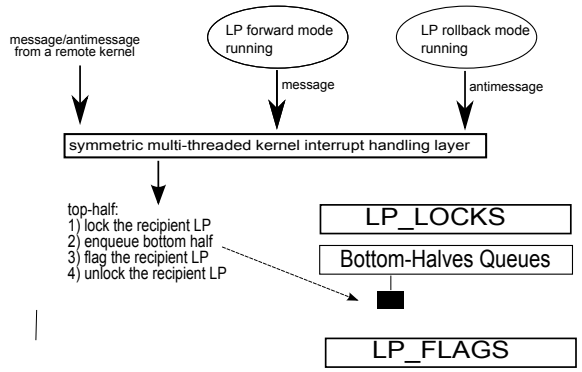

Figure 2: The top/bottom-halves architecture.

in three different circumstances:

(i) The worker thread is currently running $LP^j_{k_i}$ in forward mode, and this LP produces a new event to be scheduled for the locally hosted $LP^t_{k_i}$. Thus the worker thread enters housekeeping for actuating the delivery of the corresponding message to $LP^t_{k_i}$'s input-queue. (Note that j might be equal to t, in which case sender and receiver coincide.)

(ii) The worker thread is currently running the locally hosted $LP^j_{k_i}$ in rollback mode (hence it is performing housekeeping operations associated with revealed causality errors), which gives rise to the production of an antimessage destined to $LP^t_{k_i}$, which again requires access to $LP^t_{k_i}$'s input queue for annihilating the original message. (Also in this case we might have $j = t$.)

(iii) The message passing layer notifies the worker thread (e.g. via an explicit message receive operation executed by this thread according to a traditional polling scheme) about a new message/antimessage incoming from some remote kernel instance, which is destined to a locally hosted LP.

As shown in Figure 2, we logically mark all the above three circumstances as interrupts, which will be treated homogeneously, and whose associated message/antimessage delivery operation will be finalized via the bottom-half mechanism.

We note that spin-locks may anyhow exhibit non-minimal costs since they require the corresponding operations to be performed via sequences of atomic instructions (e.g. via the LOCK prefix for the IA-32 instruction set). Additionally, since they are shared and accessed by different threads, cross-cache invalidation effects can be induced as soon as one worker thread gains control on the spin-lock. To reduce these effects, we devise the presence of an additional array of flags `LP_FLAGS` (see again Figure 2), where `LP_FLAGS[j]` indicates whether the corresponding bottom-half queue, namely the one associated with the j-th locally hosted LP, is not empty. `LP_FLAGS[j]` gets updated within the critical section protected by `LP_LOCKS[j]`, either when a new bottom-half is inserted within the corresponding queue (in this case the flag is raised), or when the queue is flushed (in this case the flag is reset). However, `LP_FLAGS[j]` is also accessed before trying to lock the bottom-half queue in order to avoid spin-lock operations in all the cases where the queue would reveal empty once accessed within the critical section leading to flush operations. The exact scheme is:

```
TOP-HALF:
 lock(&LP_LOCKS[j]);
 <log bottom-half>;
 LP_FLAGS[j] = TRUE;
 unlock(&LP_LOCKS[j]);
```

```
BOTTOM-HALF:
 if (LP_FLAGS[j])
   if (try_lock(&LP_LOCKS[j])){
      <unlink bottom-halves>;
      LP_FLAGS[j] = FALSE;
      unlock(&LP_LOCKS[j]);
      <perform bottom-halves>;}
```

Being `LP_FLAGS[j]` checked non-atomically wrt lock acquisition when attempting to perform bottom-halves, we might experience false negatives in case the top-half finalizes the insertion of the bottom-half task concurrently with the check. However, this does not represent a safety problem since the flag will be rechecked periodically in subsequent attempts to flush the corresponding bottom-half queue, thus eventually falling in the case where the bottom-half queue is correctly reflected into the state of the input queue of the destination LP. Such a reflection might therefore experience only a delay, which resembles delays introduced by traditional single-threaded kernels while reflecting the content of cross-kernel messages into the system state, which is typically affected by the polling period according to which the messaging layer is accessed for acquiring not yet delivered messages. Further, as hinted in footnote 4, when allowing a single worker thread at a time to manage flush operations for its affine LPs, no false positives will ever be experienced.

6. EXPERIMENTAL RESULTS

6.1 Test-bed Platform

We have implemented the load sharing approach within ROOT-Sim, an open source C/MPI-based simulation package targeted at POSIX systems [12], which 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 to be implemented according to the ANSI-C standard, which represent application entry points for providing control to the LPs. Also, it transparently supports all the services required to parallelize the execution and offers a set of optimized protocols aimed at minimizing the run-time overhead by the platform, thus allowing for high performance and scalability. Among them we can mention an autonomic protocol for application transparent and performance optimized management of state log/restore operations [23].

The single threaded version of ROOT-Sim has been recently enhanced with innovative transparent supports for LP migration and load balancing [20]. This has been done via the integration of a global memory manager, allowing to manage virtual addressing on a global scale across different simulation-kernel instances. This allows allocating memory buffers belonging to the state of each individual LP such in a way to be re-mappable onto the address space of remote kernel instances, while correctly maintaining pointer-based references. Further, this architecture also offers optimized pack/unpack protocols for actual transfer operations of LPs' information across different kernels upon migrations. Overall, this is a fully featured load balancing implementation that we will consider as a reference for a comparative assessment of our current load sharing proposal.

Integration of load sharing within ROOT-Sim, according to the architectural indications provided in Section 5, has been based on `pthread` technology, and on the reorganization of housekeeping data structures in order to (i) provide per-thread private data, and (ii) cache aligned memory buffers, so to avoid false cache sharing across the worker threads within the same kernel instance. The latter objective has been achieved by exploiting the `posix_memalign`

API, plus the usage of proper padding schemes allowing cache alignment for sequences of records, such as arrays of values. As for the accesses to the MPI layer, in our architecture they can be symmetrically issued by any of the worker threads operating within a given kernel instance. Given that the MPI does not natively support multi-threading, we have included a wrapper that synchronizes these accesses transparently to the worker threads via the embedding of critical sections protected by spin-locks.

As far as GVT computation and fossil collection are concerned, we implemented a symmetric scheme in which the worker threads operating within a same kernel instance run a race. The race winner computes the local reduction and interacts with the master kernel in order to determine the globally reduced value representing the new GVT. However, once defined the new GVT value, all the worker threads operating within the same kernel instance are allowed to perform fossil collection operations in parallel, each one fossil collecting obsolete information for its affine LPs.

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.

6.2 Application Benchmarks

To evaluate different aspects of the proposed load sharing approach, and of the associated symmetric multi-threaded architecture, we have conducted experiments on two different application benchmarks, namely *Personal Communication System* (PCS) and *Traffic*, which are hereby described.

The PCS Benchmark. 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. The power regulation model has been implemented according to [14]. Also, the evolution of the state of a cell has been modeled by an individual LP.

Upon the start of a call destined to a mobile device, a call-setup record is instantiated and linked to a list of already active records within that same cell. A record is released when the corresponding call ends or is handed-off. In the latter case, a similar call-setup procedure is executed at the destination cell. Upon call setup, power regulation is performed, which involves scanning the aforementioned list to compute the minimum transmission power allowing the current call to achieve the threshold-level SIR value. Data structures keeping track of fading coefficients are also updated while scanning the list, according to a meteorological model defining climatic conditions, accounting for climatic variations with a minimum time granularity of ten seconds.

With this benchmark, we have performed a set of experiments where each wireless cell (hence each LP) sustains the same workload of incoming calls. Therefore this benchmark application has been used essentially to measure the overhead imposed by the symmetric multi-threaded organization, while not taking advantages from its ability to reallocate CPU-cores according to the load sharing policy depicted in Section 4, just given the constancy of the workload insisting on each LP. We have run experiments with 1024 wireless cells, modeled as hexagons covering a square region, each one managing 1000 wireless channels. Hence, 1024 LPs were included in each simulation run.

The Traffic Benchmark. Traffic implements a simulation model for a complex highway system (at a single-car granularity), with topology expressed as 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. Every LP is in charge of simulating a node or a portion of a segment, the length of which depends on the total highway's length and the selected number of LPs.

Cars enter the system according to an Erlang probability distribution, they can join the highway starting from cities/junctions only, and are later directed towards highway segments with uniform probability. Whenever a car is received, it is enqueued in the LP's list of traversing cars, and its speed is updated according to a Gaussian probability distribution. Then, the model computes the time the car will need to traverse the node, considering traffic slowdowns and accidents which are again computed according to a Gaussian distribution on the number of cars which are currently passing through the node/segment. When an accident occurs, the cars are not allowed to leave the LP, until the road is freed (the duration of the accident phase is derived from a Gaussian probability distribution).

We have simulated the whole Italian highway network on top of 1024 LPs. We have discarded the highways segments in the islands in order to simulate an undirected connected graph, which allows to have the actual workload migrating overall the highway. The topology has been derived from [1], and the traffic parameters have been tuned according to the measurements provided in [21]. The average speed has been set to 110 Km/h, with variance of 20 Km/h, and accident durations have been set to 1 hour, with 30 minutes variance. This model has provided results which are statistically close to the real measurements reported in [2].

6.3 Results

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 carried out. As hinted, 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 3, where all the samples have been obtained as the average over 10 runs all done with different pseudo-random seeds. One reported curve is related to a classical single-threaded execution of the ROOT-Sim simulation-kernel process, where load balancing is excluded, in which case we always have 32 kernel instances,

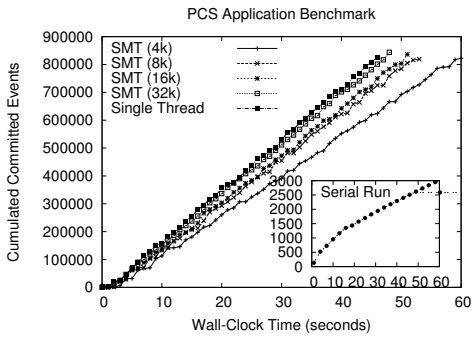

Figure 3: Cumulated event rate for PCS.

each one running on top of a different CPU-core and managing 32 out of 1024 LPs. This curve is used as the baseline for the assessment of the overhead by the symmetric multi-threaded organization, denoted as SMT in the plots.

The SMT configuration with 4 kernels (4k) shows the maximal overhead (expressed by 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 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, when implemented according to the proposed symmetric multi-threading paradigm. Also, we note that the parallel approaches provide super-linear speedup with respect to a serial run of the same identical application level code, relying on the calendar-queue [5]. This indicates how our experimentation has been carried out in the context of competitive parallel runs, thus being significant.

To further investigate on sources for the overhead, and to more deeply assess dynamics associated with the symmetric multi-threaded organization supporting load-sharing, we report additional measurements. Specifically, in order to provide quantitative data related to potential variations of the execution locality and its effects, we focused on three parameters: (a) The latency for taking a checkpoint of the LP state; (b) The latency for reloading a previously taken checkpoint in case of rollback; (c) 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 (in which case a single kernel instance is active, constantly hosting the 32 worker threads operating on top of the 32 CPU-cores available within the architecture).

By the results, shown in Figure 4, we see how the event processing cost does not show significant variations in any of the considered configurations. At the same time, the cost for taking or reloading a checkpoint provided by the symmetric multi-threaded architecture is fairly comparable to the one achieved by the single-threaded architecture for a number of kernel instances greater than or equal to 8. On the other hand, a more significant increase of the latency for taking or reloading a checkpoint is noted for SMT configurations entailing 4 or less kernel instances, where an increased number of worker threads operate within each instance. This denotes a slightly reduced locality and/or, for the case of checkpoint-taking, an increase of thread contention while accessing the malloc library for allocating buffers (whose internal synchronization for thread-safeness relies on futexes). However, taking checkpoints is typically executed infrequently (by properly optimizing the tradeoff between checkpointing and restore overheads [23]), thus leading the associated increased latency to be likely affordable. On the other hand, the frequency of the checkpoint-reload operation depends on the rollback pattern. Hence the increase of the checkpoint-reload latency is expected to acquire some relevance only in sub-optimal configurations, which may lead to frequent rollbacks.

In relation to the latter metric, we report in Figure 5 the observed values for the efficiency, namely the percentage of non-rolled back events, for all the investigated configurations. By these data we see how SMT configurations lead to efficiency values quite close to those achieved with the single-threaded architecture, except for the case where the number of kernel instances is set to 2 or 1, having initial number of worker threads set, respectively, to 16 or 32. From these and the previously shown data, we get that one major source for the overhead increase, when increasing the number of per-kernel worker threads over some threshold, resides in 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 (especially when the level of concurrency within any kernel instance gets increased, via increase of the number of worker threads within that kernel instance), 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 (rather than exclusively relying on a polling scheme, as in our current implementation), so to not induce excessive delay in the final delivery of events/antievents. We plan to investigate along this direction as future work.

Always in relation to the management of the top/bottom-half scheme within the symmetric multi-threaded architecture, we report in Figure 6 the observed per-event latency for executing top/bottom-half code blocks (excluding the final update of the input queues, which is independent of synchronization dynamics proper of top/bottom-half man-

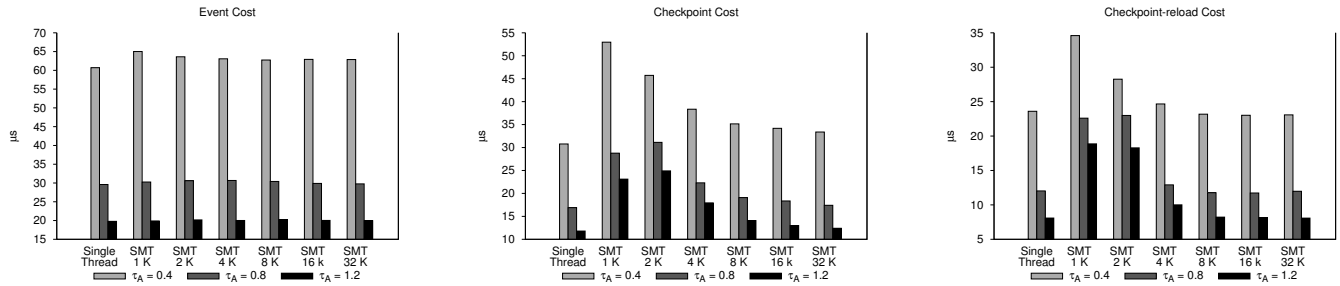

Figure 4: Costs of log/restore operations and event processing for PCS.

agement). As hinted, whenever a top/bottom-half operation must be performed by some worker thread, a lock on the bottom-half queue associated with the destination LP must be taken (although in the case of bottom-halves processing, the lock is only needed for de-queueing the events' chain currently registered within the queue). If the number of concurrent worker threads within a kernel instance grows, the contention on the queues is increased, since the worker threads synchronizing on this resource must wait for the lock to be granted to them. At the same time, since a higher number of available worker threads entails a higher number of handled LPs per-kernel instance, this statistically reduces contention on per-LP queues, so that this latency is expected to grow, but up to a certain (not large) extent. This is confirmed by the experimental results, since we see that when the number of per-kernel worker threads increases (i.e. the number of kernel instances decreases), the top/bottom-half cost increases just linearly and moderately.

The last set of data for PCS is reported in Figure 7, and is related to the latency for the management of GVT operations. As illustrated, in our symmetric multi-threading paradigms, a single worker thread (the race winner) is allowed to perform the local reduction operations on each kernel instance. This is reflected in an increase of the GVT latency when the number of worker threads per-kernel instance gets increased (i.e. when the number of kernel instances gets decreased). The only exception is for the SMT configuration with 1 kernel, in which case, exchanges of MPI-messages is avoided at all (given that a single kernel instance is active, with 32 worker threads running on the 32 available CPU-cores), which leads to avoid message passing overhead while supporting the global reduction (the GVT latency does not exhibit significant reduction for $\tau_a = 0.4$ due to the impact of the increased event granularity on the latency of the race, observable especially with higher levels of intra-kernel concurrency). Anyway, we note that the absolute values for the latency of GVT operations is quite limited, and does not represent a major source of overhead especially when considering that GVT is typically computed relatively infrequently (each 1 second in our experiments).

Concerning the results for the Traffic benchmark, which are shown in Figure 8, we have compared the cumulated event rate achieved by SMT configurations offering load sharing facilities, with the one by classical single-threaded operating mode of ROOT-Sim, with and without load balancing facilities activated, and with the one related to serial execution of the same application-level software running on top of a calendar-queue scheduler. Unlike PCS, Traffic has a highly variable load profile across the LPs, hence taking real

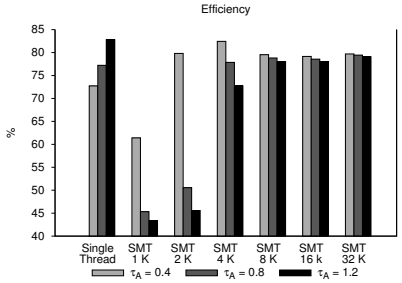

Figure 5: Efficiency values for PCS.

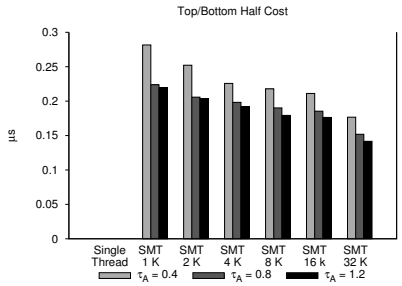

Figure 6: Top/bottom-halves costs for PCS.

advantages from dynamic resource-optimization schemes.

Again, the parallel runs provide super-linear speedup. Further, compared to the single-threaded run with no load balancing, all SMT configurations provide speedup ranging between 40% (for the 4 kernels configuration) and 55% (for the 8 and 16 kernels configuration). The execution with 4 kernel instances shows a reduced speedup due to several reasons: (i) CPU-core re-assignment 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 on the underlying NUMA machine with higher probability.

As for the execution with 32 symmetric multi-threaded kernels, the speed down is in the order of 15%. In this configuration no CPU-core re-assignment is possible (in fact, each simulation kernel must have at least one worker thread in order to proceed in the simulation run). Therefore, we are again measuring the architecture's overhead, which is indeed comparable to the one shown when running the PCS (balanced) benchmark. As for the single threaded operating mode of ROOT-Sim, with load balancing facilities activated, it provides a speedup in the order of 40% wrt the single-threaded approach without load balancing. However, while its cumulated event rate is comparable with the 4 kernels

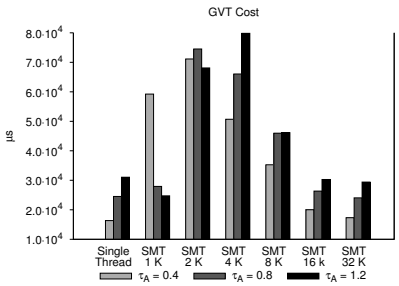

Figure 7: GVT cost for PCS.

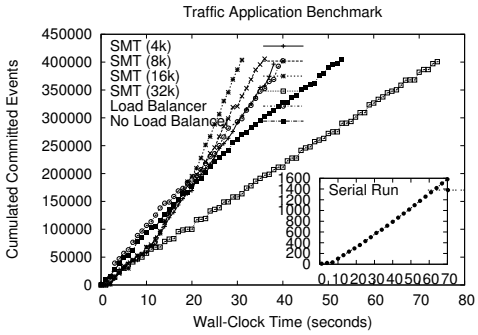

Figure 8: Cumulated event rate for Traffic.

SMT configuration, the 8 and 16 kernels SMT configurations, thanks to the offered load sharing facilities, are still 30% faster than the single-threaded configuration with load balancing. This is a clear experimental support of the effectiveness of our load sharing proposal.

7. CONCLUSIONS

In this article we have provided an innovative load sharing approach for optimizing the usage of computing resources in optimistic PDES systems run on top of multi-core machines. We have defined a load sharing model suited for dynamically determining the amount of CPU-cores to be assigned to each instance of the simulation-kernel process, in order to sustain the whole workload associated with the locally hosted LPs. A symmetric multi-threaded architectural organization has been also devised as a means for supporting load sharing in practical contexts. Further, a real implementation has been provided and experimentally assessed.

8. REFERENCES

[1] Atlante stradale italia. http://www.automap.it/.

[2] ACI. Dati e statistiche. http://www.aci.it/?id=54.

[3] Baltas, N., and Field, T. Continuous performance testing in virtual time. In *Proceedings of the 9th International Conference on Quantitative Evaluation of Systems* (2012).

[4] Boukerche, A., and Das, S. K. Dynamic load balancing strategies for conservative parallel simulations. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation* (1997), pp. 20–28.

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

[6] Carothers, C. D., and Fujimoto, R. M. Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms. *IEEE Transactions on Parallel and Distributed Systems 11*, 3 (2000), 299–317.

[7] Chen, L.-l., Lu, Y.-s., Yao, Y.-P., Peng, S.-l., and Wu, L.-d. A well-balanced time warp system on multi-core environments. In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation* (2011), PADS, IEEE Computer Society, pp. 1–9.

[8] Dantzig, G. B. Discrete-variable extremum problems. *Operational Research*, 5 (1957).

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

[10] Glazer, D. W., and Tropper, C. On process migration and load balancing in time warp. *IEEE Transactions on Parallel Distributed Systems 4*, 3 (1993), 318–327.

[11] Hamada, T., and Nitadori, K. 190 tflops astrophysical n-body simulation on a cluster of gpus. In *Proceedings of the 2010 ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis* (2010), SC, IEEE Computer Society, pp. 1–9.

[12] HPDCS Research Group. ROOT-Sim: The ROme OpTimistic Simulator - v 1.0. `http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/`, Oct. 2012.

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

[14] Kandukuri, S., and Boyd, S. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE Transactions on Wireless Communications 1*, 1 (2002), 46–55.

[15] Lin, Y.-B., and Lazowska, E. D. Processor scheduling for Time Warp parallel simulation. In *Proceedings of the 23rd SCS Multiconference on Advances in Parallel and Distributed Simulation* (Jan. 1991), IEEE Computer Society, pp. 11–14.

[16] Liu, T., Curtsinger, C., and Berger, E. D. DThreads: Efficient deterministic multithread. In *Proceedings of the 23rd Symposium on Operating Systems Principles (SOSP 2011)* (2011), ACM, pp. 327–336.

[17] Mellon, L., and West, D. Architectural optimizations to advanced distributed simulation. In *Proceedings of the 27th conference on Winter simulation* (1995), WSC, IEEE Computer Society, pp. 634–641.

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

[19] Park, A., and Fujimoto, R. M. 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* (2008).

[20] Peluso, S., Didona, D., and Quaglia, F. Application transparent migration of simulation objects with generic memory layout. In *Proceedings of the 25th Workshop on Principles of Advanced and Distributed Simulation* (june 2011), IEEE Computer Society, pp. 169–177.

[21] per L'Italia S.p.A., A. Reportistica sul traffico. `http://www.autostrade.it/studi/studi\_traffico.html`.

[22] Quaglia, F., and Cortellessa, V. On the processor scheduling problem in time warp synchronization. *ACM Transactions on Modeling and Computer Simulation 12*, 3 (July 2002), 143–175.

[23] Vitali, R., Pellegrini, A., and Quaglia, F. Autonomic log/restore for advanced optimistic simulation systems. In *Proceedings of the Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (2010), IEEE Computer Society, pp. 319–327.

[24] Weatherly, R. M., Wilson, A. L., Canova, B. S., Page, E. H., Zabek, A. A., and Fischer, M. C. Advanced distributed simulation through the aggregate level simulation protocol. In *HICSS (1)* (1996), pp. 407–415.