Granular Time Warp Objects

Nazzareno Marziale, Francesco Nobilia, Alessandro Pellegrini and Francesco Quaglia
nazzareno.marziale@gmail.com, f.nobilia@gmail.com, pellegrini@dis.uniroma1.it, quaglia@dis.uniroma1.it
DIAG – Sapienza, University of Rome

ABSTRACT

A recent trend has shown the relevance of PDES paradigms where simulation objects are no longer seen as fully disjoint entities only interacting via events' scheduling. Particularly, mutual cross-state access (as a form of state sharing) can represent an approach enabling the simplification of the programmer's job. In this article, we present a multi-core oriented Time Warp platform supporting so called *granular objects*, where cross-state access is transparently enabled jointly with the dynamic clustering (granulation) of objects into groups depending on the volume of mutual state accesses along phases of the model execution. Each group represents an island where activities are sequentially dispatched in timestamp order. Concurrency is still preserved by enabling the optimistic execution of the different islands. Granulated objects do not pay synchronization costs due to mutual causal inconsistencies. Also, the underlying Time Warp platform does not pay memory management (e.g. memory access tracing) overheads to determine that mutual accesses are taking place within a group. Overall, the platform transparently (and dynamically) determines a well-suited granulation of the overall model state, and a corresponding level of concurrency, depending on the actual state access pattern by the simulation code. As far as we know, this is the first study where the problem of clustering Time Warp simulation objects is addressed for the case of in-place cross-object state accesses by the application code, and where dynamic granulation of multiple objects in a larger one is supported in a fully transparent manner. We integrated our proposal in the open source ROOT-Sim platform.

Categories and Subject Descriptors

I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete Event, Parallel*

Keywords

PDES; Multi-thread; Linux; Kernel Support; Synchronization

SIGSIM-PADS '16, May 15-18, 2016, Banff, AB, Canada

DOI: http://dx.doi.org/10.1145/2901378.2901390

1. INTRODUCTION

Historically, Parallel Discrete Event Simulation (PDES) [9] has been based on a programming paradigm where (i) the overall simulation model state is partitioned into disjoint sub-states, each one associated with a so called simulation object (or Logical Process—LP), (ii) the memory access operations by the event-handling routine are confined within the state of the LP targeted by the event, and (iii) interactions across the LPs solely occur by the exchange of messages carrying *timestamped* simulation events.

Although this paradigm fully fits loosely-coupled parallel architectures, where LPs are executed concurrently on distributed memory systems, the sliding towards multi/many-core off-the-shelf technology has renewed challenges and opportunities in the design of shared memory oriented PDES environments (see, e.g., [6, 23, 24]).

This aspect also involves the development of innovative programming paradigms, and their associated runtime support, where the application software is no longer limited to perform per-event state access/update operations in data separation across the LPs. Rather, state-wide in-place access by event-handlers, exploiting shared memory in the underlying multi-core machine, is nowadays considered as a powerful means for improving programmability of complex models and further increasing model execution speedup [16]. Just to provide a few examples, it can avoid the coding of complex cross-LP event patterns in case some data embedded within the state of a given LP need to be known while processing an event occurring at some other LP. Also, for (very) large models, where such data awareness would involve large amounts of LPs, the possibility to directly access the target data in-place while coding the event-handlers for whatever LP would also lead to avoiding large message-exchange overhead.

Following this research trend, in this article we present the design of a multi-core oriented Time Warp PDES environment supporting so called Granular LPs (GLPs). Our proposal is based on enabling in-place access to the state of whatever LP by the event-handlers, particularly by inheriting the basic mechanisms already offered by the support presented in [16]. However, differently from such proposal, we also enable the dynamic grouping of multiple LPs, thus forming islands that are managed by the runtime PDES environment as if the modeler had specified them to represent a same (larger) object within the whole simulation model.

Overall, while we support the same PDES programming model as in [16], the granulation process leads to some major differences in the runtime behavior of the underlying Time Warp system. On the one hand, a GLP does not

pay any rollback-related synchronization cost due to mutual causal inconsistencies of its member LPs, given that all of them are always scheduled sequentially and according to timestamp ordering of the events they receive as input. On the other hand, once a GLP has been dynamically formed, runtime tracing/identification of memory operations by event-handlers, which indiscriminately access the states of its members, does not need to be carried out, thus avoiding the associated costs. This occurs because GLP members do never execute concurrently, thus any event-handler accessing the overall GLP state always observes a consistent (although speculatively-generated) snapshot. Contrarily, the absence of granulation in [16] led this proposal to rely on simulation-lifetime cross-state access tracing (via an ad-hoc Linux memory management support). This is exploited to apply an explicit LPs' synchronization protocol enabling consistent snapshot access when multiple LPs' states are touched by in-place memory operations by some event-handler.

In our proposal, concurrency and the potential for exploiting model parallelism are still preserved by keeping the different GLPs—or individual LPs not involved in the dynamic granulation process—executing according to optimistic synchronization. Also, a GLP is dynamically formed only in case the runtime support determines that a group of LPs exhibits (along a specific model execution phase) significant cross-state dependencies, caused by repeated in-place accesses to their states while processing individual events. This might be the case of, e.g., an agent-based simulation model where some LP represents an individual, and other LPs represent a portion of the environment, and at a certain point multiple individuals start interacting repeatedly with the same portion of the environment—a scenario just coded via in-place access to the state of multiple LPs by some event-handler. If this is no longer the case, the GLP is un-grouped so as to slide along classical concurrent speculative execution of its members. The dynamic granulation process is driven by a self-adapting policy, which is used to determine how long a GLP should exist, given that it is expected to provide some revenue in terms of synchronization efficiency. Our policy is based on a mixture of space-efficient probabilistic data structures and machine learning.

As hinted, our proposal is specifically designed to reduce both direct (e.g., due to speculation) and indirect (e.g., due to memory access tracing) synchronization costs, as compared to the original cross-state synchronization approach in [16]. Further, it targets the reduction of the overhead when LPs are tied/untied to a given GLP. To this end, a GLP has multiple input/output queues, one for each LP belonging to it. This poses algorithmic challenges, and requires, as it will be shown, to redesign as well traditional operations characterizing the execution of a Time Warp-based simulation, such as the *schedule*, the *rollback*, and the *coasting forward* phase (in case of checkpoint-based rollback).

The remainder of this article is structured as follows. An overview of the cross-state synchronization approach is given in Section 2, so as to summarize the results from [16] which are exploited in the current proposal. In Section 3 we discuss the innovative architectural organization of a Time Warp system supporting the granulation process, and the algorithms it uses. An assessment of validity and effectiveness of our proposal is presented in Section 4. In Section 5 we discuss related work.

2. CROSS-STATE SYNCHRONIZATION RECAP

The proposal in [16] allows the event-handler of DES models developed in `C` technology to access (and alter) the state of multiple LPs while processing any individual event. Therefore it provides and enhanced support for model development, which does not only rely on cross-LP scheduling of events in order to code the dependencies across the different model's portions. Rather, in-place read/write operations are enabled targeting any (dynamic) memory location that represents a valid buffer included in the state of whichever concurrent LP. This type of operation is referred in [16] to as *cross-state access*.

Cross-state accesses must be supported in such a way to ensure that the state snapshot observed by the event-handler is consistent, although generated by a speculative execution. Hence, the LPs whose states are actually accessed while processing an individual event all need to figure as aligned (in logical time) to the timestamp of the event. This is achieved by encapsulating the cross-state access within an atomic action that is, in its turn, based on an ad-hoc synchronization protocol triggered on demand, if and only if a cross-state access materializes. Such a materialization is detected in [16] by relying on an advanced Linux oriented virtual memory management architecture able to track at runtime the access to the state of whichever LP.

The architecture delivers memory buffers (in reply to `malloc` requests by the application code) via a non-anonymous scheme, where the buffers destined to serve memory requests by an LP are guaranteed to fall within a memory *stock* formed by a 1GB-wide segment of contiguous virtual addresses just reserved for hosting the state of that LP. Hence, with stock-based allocation, the virtual memory pages destined to host memory buffers included in the state of an LP correspond to contiguous pages whose virtual-to-physical memory translation is associated with a single entry of the second-level x86-64 page table, which is called PDP—Page Directory Pointer.

A special device file (whose driver is loaded into the Linux kernel via an external module) is used to communicate via the `SET_VM_RANGE` `ioctl` command, implemented within the module, what is the range of virtual addresses associated with a given simulation object. Before CPU-dispatching an event at some LP, the worker thread in charge of the execution communicates to the kernel module what is the LP which will be activated via another `ioctl` command called `SCHEDULE_ON_PGD`. This command activates a kernel-level logic implemented in the module which installs a *sibling page table* on the `CR3` (page-table pointer) register of the CPU-core running the worker thread. The per-thread sibling page table is constructed so that initially the thread has access only to the pages destined to keep the dispatched LP state—trying to access any other LP state while processing the event generates a fault.

The fault is efficiently intercepted by the kernel module, which gives control back to the worker thread (thus, outside of the execution context of the LP, thanks to ULT support). In this way, the execution of the current event—hence of the associated LP—is interrupted, and it could be eventually restarted without worrying about any housekeeping operation which took place in the meanwhile[1]. In

[1]We recall that the LPs' execution contexts—including their stacks—are separated in this system.

addition to blocking the execution of the LP, the worker thread sends a *rendez-vous start* control message towards the LP whose state is the target of the intercepted memory access (cross-state) operation. Such control messages carry system-events that are silently (with respect to LPs' activities) exchanged by worker threads. System-events do not cause the actual dispatching of the target LPs for state updates, although they are incorporated into the LPs' event lists as place-marks. Independently of whether the rendez-vous start event is located in the past or future of the target LP logical time, this LP will eventually re-align its logical time to the rendez-vous timestamp either via a rollback operation, if the rendez-vous start event was a straggler, or by executing events until the rendez-vous is reached.

When an LP reaches a rendez-vous start event in logical time, it is temporarily blocked, and the Time Warp platform sends back to the LP originating the cross-state access a rendez-vous ack control message, which puts this LP back to the *ready* state. In this way, the event that was temporarily blocked due to cross-state access interception could be restarted from the machine instruction which caused the fault. Nevertheless, this time, when the `SCHEDULE_ON_PGD` command is executed before reactivating the LP, the kernel module will be instructed to open the access to the memory stocks of both the LPs involved in the cross-state access, thus allowing in-place memory access by the event-handler to both their states. This scenario could be repeated more than once per each event, thus synchronizing multiple LPs and opening the access to all their states. When the execution of an event that caused a cross-state synchronization is completed, the worker thread sends to all the LPs that were involved in the synchronization a *rendez-vous unblock* control message, which brings all them back into the ready state, for normal processing.

By the above description, the materialization of a cross-state access leads to a non-persistent relation between two or more LPs. In fact, given that cross-state synchronization is operated on a per-event basis, after the finalization of the event that led to cross-state accesses, the involved LPs start again executing alone along their own simulation trajectories. However, in general contexts, a cross-state access by the application code could be the evidence that two (or more) LPs are actually starting to execute in a synergistic way, in terms of overall simulation model execution trajectory. In this scenario, the cross-state synchronization protocol as introduced in [16] might require the exchange of a large number of control messages—which in turn might produce a large amount of rollbacks as well. Our innovative LPs' granulation process aims at avoiding adverse runtime synchronization dynamics associated with such a synergic execution of the LPs, while still supporting the flexible programming model offering the possibility to perform in-place accesses to whatever state information by the event-handler.

3. GRANULAR LOGICAL PROCESSES

3.1 The Grouping Algorithm

Our support for instantiating and managing GLPs inherits the memory management architecture presented in [16]. However, the materialization of cross-state accesses by the application software, beyond being used for synchronizing the involved LPs, is also used to determine a relation across them, which will drive the formation (or fading) of GLPs.

As hinted, we consider the fact that two LPs are involved in a cross-state synchronization as a possible materialization of a synergistic execution phase. To keep track of this, we rely on a matrix $LpDependencies$, the size of which is $numLPs \times numLPs$. Every time that a cross-state access involving LP_i and LP_j is detected at runtime according to the technique described in Section 2, we increment the value of the two matrix elements $LpDependencies[i,j]$ and $LpDependencies[j,i]$. $LpDependencies$ represents therefore a matrix of cross-state dependency counters, where every element (i,j) tells how many cross-state dependencies involving LP_i and LP_j have been detected along some execution phase[2].

The $LpDependencies$ matrix can be mapped to an incidence matrix of a directed multigraph $G = (V, E)$ where the set of vertices V keeps the identifiers of all the LPs currently running in the system, and the set of edges E is defined as $E = \{\{i,j\} : i,j \in V \wedge LpDependencies[i,j] > 0\}$. Nevertheless, before converting it to an actual incidence matrix, we apply some filtering aimed at reducing the possibility of capturing spurious cross-state relations (depending on the events' logic). This is because the GLP concept targets the runtime optimization achievable in scenarios where actual synergistic execution due to cross-state accesses is statistically significant. Hence, cross-state relations which are detected only seldom should not be taken into account for driving the dynamic formation of GLPs. Overall, we use a threshold τ_{dep} to filter out all the spurious cross-state dependencies, thus building a so called *cross-state dependency multigraph* $G = \{\{i,j\} : i,j \in V \wedge LpDependencies[i,j] \geq \tau_{dep}\}$ and deriving its incidence matrix which we refer to as IMG. If no edge exists in G between two LPs, say LP_i and LP_j, then we set the value of the IMG element (i,j) to the special value $\perp$. These data structures are exploited to determine the formation of GLPs according to the following scheme.

Periodically (say after n cycles of GVT computation) IMG is accessed to determine what is the highest value of cross-state access counts for each LP_k thus determining the index:

$$MaxDep_k = \max_{i \in [0, numLPs-1], i \neq k} \{IMG[k,i]\}$$

where the element $\perp$ is assumed to be the lowest value in the domain where the maximum is searched. These indices are used to build a so called $LpGranulation$ vector, which is a vector of tuples each one structured as $\langle MaxDep_k, group \rangle$ $\forall k \in [0, numLPs - 1]$. Initially, the value $group$ for all the elements of the vector is set to the special value $\perp$. This configuration tells that LP_k, associated with the k-th row of the $LpGranulation$ vector, has its highest dependency counter set to $MaxDep_k$ and belongs to the special group $\perp$, meaning that LP_k still belongs to no group.

This construction transforms the multigraph G into another oriented multigraph $\bar{G}$ such that the set $\bar{V} \equiv V$, but if $\{i,j\} \in \bar{V}$, then $\{i,k\} \notin \bar{V} \quad \forall k \neq j$. This means that every node $i \in \bar{V}$ has at most one edge connecting it to another node $j \in \bar{V}$, with $i \neq j$, and by construction $j = MaxDep_i$.

[2]This matrix is a logical construction, while the actual counters have been embedded in our implementation within LP control blocks, which keep recoverable information. Consequently, updates if the matrix entries associated with rolled back computation are discarded.

A graph visiting algorithm on $\bar{G}$ is then used to determine the formation of GLPs as groups of LPs[3]. We iterate over all indices $k \in [0, numLPs - 1]$, and for each value k we execute the recursive function REGROUP(*LpGranulation*, k, $\perp$) shown in Algorithm 1. The goal of this recursive function is to determine whether the selected LP already belongs to a group. In the negative case, if the passed value for the group ID is not $\perp$, then the target LP is aggregated into the passed group (line 6), otherwise a new group is created, which is associated with the ID of the passed LP (line 8). In the positive case, no action is taken for the current LP, and the group the LP belongs to is returned (line 3). Both cases (namely, lines 6 and 8), are associated with *tentative groups*, which could be later confirmed or discarded. If the LP was associated with a tentative group, a recursive call is issued to REGROUP() (line 11), selecting as the target LP the $MaxDep$ one of the current LP, and passing the ID of the group which the current LP belongs to. The group ID of the current LP is then updated with the return value of this call, which is done so as to backwards propagate the creation of new groups or the agglomeration to existing ones (line 13). Line 11 can either confirm a tentative group for a given LP, or supersede it with a different one.

Algorithm 1 GLP Construction

```
1: procedure REGROUP(LpGranulation GLP, int LPid, int group)
2:     if GLP[LPid].group ≠⊥ then
3:         return GLP[LPid].group
4:     end if
5:     if group ≠⊥ then
6:         GLP[LPid].group ← group
7:     else
8:         GLP[LPid].group ← LPid
9:     end if
10:    if GLP[LPid].MaxDep ≠⊥ then
11:        GLP[LPid].group = REGROUP(GLP, GLP[LPid].MaxDep,
       GLP[LPid].group)
12:    end if
13:    return GLP[LPid].group
14: end procedure
```

We show in Figure 1 an example execution of Algorithm 1 for a scenario with 8 LPs. In the example, LP_0 exhibits a large number of cross-state dependencies involving LP_3, LP_1 shows no cross-state dependency, LP_2 is dependent on LP_6, LP_3 has no dependencies, LP_4 depends on LP_1, LP_5 depends on LP_6, LP_6 depends on LP_4.

Algorithm 1 is first invoked on LP_0, which belongs to no group (i.e., the *group* field of row 0 of *LpGranulation* is set to $\perp$), and therefore a new group with ID set to 0 is created (line 8). Then, since $MaxDep_0 = 3$, line 11 is executed as REGROUP(*LpGranulation*, 3, 0). Therefore, for LP_3, the group is set to 0 (line 5), and the value 0 is returned again at line 13, confirming the tentative group. Thus, LP_0 and LP_3 now both belong to group 0, say to a same GLP. The execution then selects LP_1 which does not belong to any group: a new group with ID set to 1 is created. Then, LP_2 is selected, which is the most interesting execution case of this example. First, this LP is set to tentative group 2 (line 8), and then the graph visiting selects LP_6. Since LP_6 belongs to no group, the new tentative group with ID 6 is created, and the visiting goes to LP_4, leading to the creation of a new tentative group with ID 4. When the visit reaches LP_1, line 3 is executed, as LP_1 already belongs to group 1. Therefore, all tentative groups for LPs 4, 6, and 2 are backwards superseded by group 1 (as per lines 11 and 13). The actual execution for LPs 3 to 7 can be trivially deduced from the already analyzed execution steps. It is interesting to note that LP_7 belongs to a group formed by a single LP.

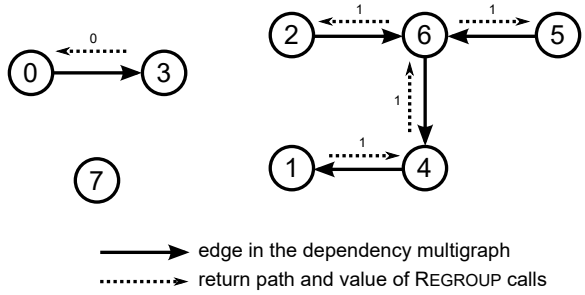

Figure 1: REGROUP **execution with 8 LPs.**

Once the graph visiting algorithm is completed, every LP belongs to a group. We note that in the scenario where no dependencies at all were detected (namely, all the elements in *LpDependencies* are found to be set to a value smaller than the threshold value τ_{dep}), Algorithm 1 creates $numLPs$ groups, each one keeping a single LP. In this case, our GLP scheme boils down to a traditional Time Warp execution, although augmented with cross-state synchronization support according to [16].

Given that groups are defined after an observation period where the materialization of cross-state dependencies has been tracked, a scenario where such cross-state dependencies, although playing a role in the construction of the G graph, were spanned over no longer actual execution phases could arise. In other words, the locality of the dependencies might vary over time according to a non-uniform distribution leading to the identification of synergistic executions no longer in place at a given point in time.

In order to capture this scenario, we have coupled the *LpDependencies* matrix with a matrix keeping timestamps values, called $TimeDependencies$[4], which records at the element (i, j) the latest wall-clock time at which the corresponding counter in *LpDependencies* was incremented. Thus, we resort to an additional threshold called $\tau_{freshness}$ which is used when building the incidence matrix for the dependency multigraph G. In particular, if a given counter element (i, j) is higher than τ_{dep}, then an edge (i, j) is added to E if, and only if, $CurrentTime - TimeDependencies[i, j] < \tau_{freshness}$. In this way, stale interactions are filtered out.

By the algorithm structure, we guarantee that if a GLP is formed, then it contains sets of LPs whose cross-state dependencies have been (recently) observed to be more intense than the ones observed for LPs that belong to different groups.

Clearly, GLPs as defined by the presented algorithm are abstractions, and need to be finally managed by the runtime system according to specific rules which we discuss in the next section. The discussion and the proposed GLPs' management approach will also cope with scenarios of underparallelism, where less GLPs than worked threads would be generated (as a corner case) by blindly running the presented

[3]From now on we will use the terms GLP and group, or GLP ID and group ID, interchangeably.

[4]This second matrix is still managed so as to make its entries recoverable in rollback phases.

grouping algorithm without taking into account the available level of parallelism in the underlying PDES platform.

3.2 Management of GLPs

After the graph visiting algorithm is completed, we consider the groups as *determined*, but not yet active—we say that the groups are not *revealed* yet. Revealing a group requires some more actions to be taken since the execution of the LPs is asynchronous, just like prescribed by the Time Warp algorithm. Hence we need to put the LPs forming a determined GLP on phase. This task is carried out according to the following scheme.

Once a GLP, say $\mathcal{G}$, is determined, the input queues of all the LPs belonging to it are examined. We recall that every LP is associated with a last (speculatively) processed event[5], referred to as bound event, which is the last correctly executed event in the speculative portion of the simulation trajectory of that LP. For each $LP_k \in \mathcal{G}$, the event e_k, associated with timestamp T_{e_k} which is the event next to LP_k's bound is selected[6]. From this pool of events, we determine the *group revelation bound* as the event $\hat{e}$ such that $T_{\hat{e}} = \max_{k \in \mathcal{G}}\{e_k\}$. Once this event—which is the next event farthest in the simulation time future—is executed at the corresponding LP, we consider the group as *revealed*.

Nevertheless, once a GLP is determined, some preliminary steps are immediately taken by the worker threads in order to CPU-schedule the events occurring at the GLP members. In other words, every operation carried out by worker threads somehow considers groups of LPs, rather than individual LPs. This is achieved by letting each worker thread receive a temporary binding of a set of LP groups, depending on the total workload of the LPs in the groups, according to the load-sharing scheme proposed in [22], which we inherit in our Time Warp implementation supporting GLPs.

Assuming, as commonly done in literature, that the CPU-scheduling operation by the worker thread works according to the Smallest-Timestamp-First (STF) policy, all the LPs in any GLP are guaranteed to execute their events in a timestamp ordered fashion (since they are all bound to a single worker thread). We can therefore augment the notion of a group $\mathcal{G}$ by introducing the *group bound*, which is the last processed event by any LP_k belonging to $\mathcal{G}$. The group bound advances whenever any LP executes an event, and goes back whenever a straggler event is received and a rollback operation is required involving elements in $\mathcal{G}$. When a group is determined (but not yet revealed) we set the group bound to the oldest LP bound in the group. When an event is executed at a certain LP of a group, both the LP bound and the group bound advance to the next event.

As mentioned before, a group is revealed whenever $\hat{e}$ is executed. With the notion of group bound, this means that a group is revealed when the group bound corresponds to $\hat{e}$. At this point the members of the group do no more execute as individual entities, rather as a single entity, say the GLP they form. Referring to the cross-state access model described in Section 2, from a technical point of view this means that upon the schedule operation of any LP in the GLP, the `SCHEDULE_ON_PGD` command receives as input the IDs of all the LPs belonging to the corresponding group,

[5]This might correspond to the INIT event of the simulation in case no model specific event was ever scheduled for that LP.

[6]If LP_k has no event next to the bound, we consider its bound.

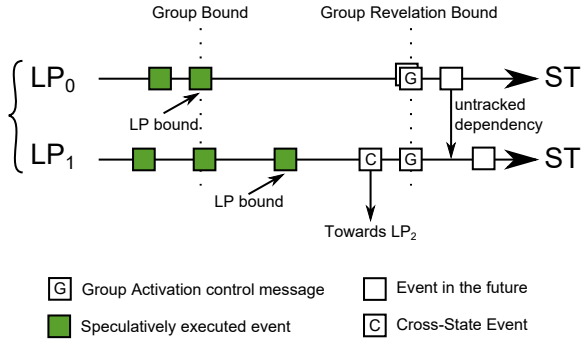

Figure 2: Revelation of a group: need for synchronization.

so that the states of all of them can be accessed without the need for tracing memory accesses and synchronizing the execution (given that the snapshot associated with the union of these states has a logical time compliant with the timestamp of the CPU-scheduled event destined to some GLP member—although the snapshot is generated speculatively). In case of contemporary events within a GLP, any tie-breaking function (see, e.g., [13]) could be applied at GLP level.

A particular case might arise, related to the revelation of a group. Consider the scenario depicted in Figure 2. A group is composed of two LPs, namely LP_0 and LP_1, whose next events are at times $T_0 > T_1$. Therefore, $\hat{e}$ is selected as the next event of LP_0 at time T_0. Since the group is determined but not revealed yet, the STF-based scheduler selects LP_1 for execution. The next event generates a cross-state dependency, which is still tracked (we recall that, until a group is not revealed, the forward execution phase is similar to the traditional Time Warp with cross-state synchronization). This cross-state dependency is materialized towards some LP which is outside the group, say LP_2, and there is no limit in the amount of wall-clock time to wait for LP_2 to send a rendez-vous ack control message. In the meanwhile the speculative execution continues, and LP_0 is selected for event processing (we recall that the LP blocked for synchronization is temporarily skipped by the STF-based scheduling policy, as described in [16]). LP_0 executes $\hat{e}$, thus the group is revealed and the execution starts according to the GLP logic: any LP has access to all the states of the LPs belonging to the group. LP_1 is still blocked waiting for the rendezvous ack control message. While processing its next event, LP_0 generates an untracked dependency, namely it accesses LP_1's state with no detection by the cross-state access tracking system. While this is the goal of the GLP abstraction, this execution is incorrect since LP_1 has not completed the execution of its event (it's blocked waiting for a rendez-vous ack control message), and therefore the memory access by LP_0 does not see a consistent state.

To overcome this issue, we augment the logic associated with the determination of the group revelation bound. In particular, after having selected $\hat{e}$, a *group-activation control message* is sent towards all the LPs in the group. Additionally, each group is associated with a *group state*, part of the *group control block*[7]. This group state is associated with a

[7]The *group control block* is a per-group data structure which keeps all the control information related to it—we will fill

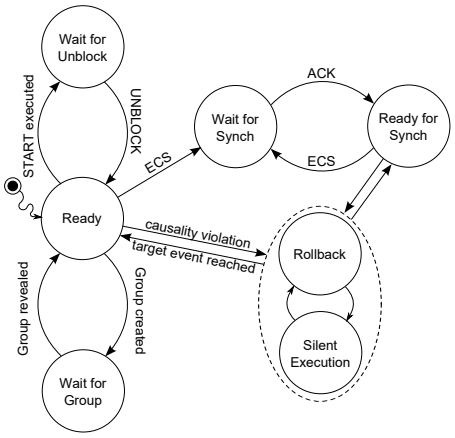

Figure 3: State machine for groups.

state machine which, as we will see later, determines the execution cycle of a group. Once a group is determined, it starts in the *wait-for-group* state. We note as well that the corner case where all LPs in a determined group have no event next to the bound leads to selectling as the group-revelation bound the timestamp associated with the bound event farthest in the simulation time. This wait-for-group state is associated with an *activation-bound counter*, again placed in the group control block and initially set to the number of LPs in the group. Whenever some LP belonging to the group reaches the group-activation control message, the simulation kernel moves it to the block state (so that no other event is processed by it until the group is revealed), and the activation-bound counter is decreased. Thus, if this counter reaches zero, it means that all the LPs in the GLP are synchronized to the revelation bound. At this point, the worker thread sets the group and all its LPs as ready. From now on, the scheduler will rely on the group state to drive the activities of the group, hence of the GLP. In particular, whenever a cross-state dependency is materialized towards some LP belonging to a different group, the whole GLP goes into the wait-for-synch state. Similarly, all the other states characterizing the original cross-state synchronization protocol in [16] are mapped to the GLP state. The state machine for a group is shown in Figure 3.

The overall logic to compute and install groups is reported in Algorithm 2. For the sake of performance, it is devised mostly as a non-blocking algorithm [11], except for a final synchronization point where all the worker threads hit a barrier. The idea behind this algorithm is that while one thread (which we refer to as the *master thread*, as the check at line 6 indicates) computes the new groups, it is pointless for the other worker threads to wait for this task to complete. Rather they can keep on processing simulation events. To let other worker threads continue processing, we rely on counters which described the *group determination era*, namely a global shared counter ($group\_era$) and a set of thread-private counters (my_group_era). When the master thread starts computing the new GLPs, the new group control blocks (which keep as well pointers to LP-related information) are computed on a global data structure. When the GLPs are all determined, via repeated calls to REGROUP(), the global $group\_era$ counter is incremented (line 10). At every main simulation loop cycle, all the other worker threads compare the value of $group\_era$ with their private value of my_group_era (line 16), and only if it is incremented they start installing the groups (line 20, for other worker threads). In this way, even though the GLPs' determination procedure takes a bit of time, the speculative execution can continue at other worker threads. Installing groups requires all the worker threads to make a private copy of the shared group control blocks on thread-private storage, and rebinding the LPs belonging to their groups on them. This algorithm requires an additional modification, related to the determination of $\hat{e}$. In fact, this should be done by the worker threads only after that the GLPs have been bound to them (lines 14 and 22). The thread barrier at the end of the algorithm is required so as to ensure that when the new GLPs are determined, no worker thread restarts executing events before all the worker threads have a coherent view on what LPs they are in charge of CPU-scheduling.

this data structure with the relevant fields during the forthcoming discussion.

Algorithm 2 GLP Management Algorithm

```
1:  new_LP_groups[]
2:  LP_groups[] (thread-private)
3:  group_era
4:  my_group_era (thread-private)
5:  procedure GroupCompute( )
6:      if MasterThread( ) then
7:          for i ∈ [0, numLPs − 1] do
8:              Regroup(LpGranulation, i, ⊥)
9:          end for
10:         group_era ← group_era + 1
11:         SanitizeGroups( )
12:         InstallGroup( )
13:         ThreadBarrier( )
14:         Compute Group Revelation Bounds for all bound groups
15:     else
16:         if my_group_era == group_era then
17:             return
18:         end if
19:         my_group_era ← group_era
20:         InstallGroup( )
21:         ThreadBarrier( )
22:         Compute Group Revelation Bounds for all bound groups
23:     end if
24: end procedure
```

An additional operation is required for avoiding under-parallelism in the model execution. In particular, it could be the case that due to the detected cross-state dependencies the number of determined GLPs according to the algorithm devised in Section 3.1 is smaller that the total number of available worker threads. To avoid under-parallelism scenarios, the master thread executes the SANITIZEGROUPS() routine (line 11), which takes care of this. In particular, if the number of GLPs $\mathcal{NG}$ is smaller than the available number of worker threads C, additional $C - \mathcal{NG}$ groups are instantiated. To determine which LPs should fall into these new groups, the *LpDependencies* matrix is scanned, so as to determine what are the LPs with the smallest number of interactions, and they are taken out of their determined groups according to a greedy approach.

An alternative approach to cope with discrepancies between C and $\mathcal{NG}$ would be the one of dynamically shrinking the amount of available worker threads. This would not be a means to avoid under-parallelism with respect to the amount of worker threads (e.g. the amount of CPU-cores potentially available within the hardware platform), rather for driving the level of parallelism (hence the actual synchronization dynamics) exclusively on the basis of cross-state dependen-

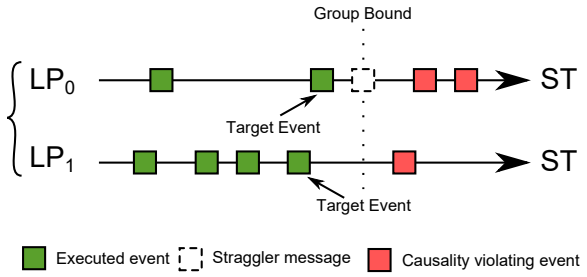

Figure 4: Group rollback.

cies materialization, according to the grouping algorithm we presented. We plan to explore this approach as future work.

3.3 Managing Rollbacks

Concerning the reception of straggler messages, the execution of the rollback operation must take into account GLPs, rather than single LPs. In fact, a straggler must rollback the entire GLP, as shown in Figure 4. In particular, in the depicted example, LP_0 receives a straggler message keeping event e at timestamp T_e, which is smaller than the bound of both LP_0 and LP_1. We note that rolling back as well LP_1 in this case is not an option. In fact, nothing prevents the received straggler to manifest a cross-state dependency when executed in forward mode. Nevertheless, since both LPs belong to the same group, LP_0 has direct access to LP_1's simulation state. Therefore, if LP_1 is not rolled back as well, LP_0 would see an inconsistent snapshot, related to a simulation time in the future.

Whenever a straggler message e_{str} is received, associated with timestamp T_{str}, the GLP's state is set to the *rollback* state, meaning that a causality violation was detected. At this point, all the LPs that are members of the GLP must rollback to a previous consistent simulation state. The first task is therefore to identify such a consistent state. If we consider a GLP $\mathcal{G}$ composed of $|\mathcal{G}|$ LPs, we select from each $LP_i, i \in [0, |\mathcal{G}-1|]$ the *target event* $\bar{e}_i$ associated with a timestamp $T_{\bar{e}_i} < T_{str}$. A *group-consistent* simulation state for $\mathcal{G}$ is the state $\bar{S}_{\mathcal{G}} = \bigcup_{i=0}^{|\mathcal{G}|-1} S_i$ such that each S_i keeps the effects of the execution of all the events in the i-th input queue up to the execution of $\bar{e}_i$, included.

The goal of the rollback operation is thus to realign the LP bound of every $LP_i, i \in [0, |\mathcal{G}-1|]$ to $\bar{e}_i$. In case the simulation engine bases its rollback operation upon checkpoint/restore primitives, it would appear sufficient to select at each LP_i the simulation state checkpoint S_i^t such that $t < T_{str}$, and then execute the coasting forward phase. Unfortunately, this simple solution could be easily proven wrong in a twofold way. Consider, in fact, the example shown in Figure 5, where GLP $\mathcal{G}$ is already revealed. The first anomaly is related to the way the coasting forward phase is usually carried out. In fact, according to [12], the traditional coasting forward phase leads the LP to silently reprocess events until the simulation state is realigned to the desired point. Referring to Figure 5, this means that LP_0 is selected, its checkpoint is restored, and events e_1, e_2, and e_3 are reprocessed. Then, LP_1 is selected, its state is restored as well, and events e_4, e_5, and e_6 are silently replayed. The problem is related to the fact that e_5 generates a cross-state dependency towards LP_0. Since the group is already revealed, dependency detection is disabled across the two involved LPs, and LP_1 directly accesses LP_0's state. Nevertheless, since LP_0 has already reprocessed e_3, LP_1 observes an inconsistent snapshot, in a way which goes undetected. The second anomaly is similar in spirit, but is actually related to the replay of e_1 at LP_0. In fact, e_1 generates a cross-state dependency towards LP_1, but: i) LP_1 has not yet restored its previous checkpoint, and ii) even if the checkpoint were already restored, it would be associated with a simulation state farther in simulation time.

To solve these inconsistencies, two changes to the traditional Time Warp operations should be actuated. First, a *group-log* operation should be supported in case of checkpoint-based rollback of GLPs, which is depicted in Figure 6. In particular, the checkpointing period χ should be evaluated at the GLP level, so that any data structure required to keep track of how many events have been processed since the last log should be kept in the group control block. Once the checkpointing system detects that a state checkpoint should be taken, a copy of the LP's state at which the last event was executed is immediately taken. Additionally, a *checkpoint control message* is sent towards all the other LPs in the group. Once the other LPs in the group are scheduled, this control message is found in the input queue, and the simulation kernel immediately takes a checkpoint state. The control message can be immediately discarded, as checkpoints are ephemeral and do not survive a rollback operation to a lower logical time—therefore a checkpoint control message should be never replayed during a coasting forward phase. In this way, whenever a checkpoint is selected during a rollback operation, it is guaranteed that any other LP in a given GLP has a checkpoint associated with the same simulation time instant, thus preventing the above second anomaly to occur[8]. This is a necessary condition in case of a rollback, and must be ensured as well in the case the group has just been determined. To this end, the execution of a group-activation control message involves as well taking a state snapshot.

As for the first anomaly, it arises from the fact that a GLP must be considered as a unique object, although associated with multiple input queues (the input queues of its member LPs). Therefore, the anomaly can be tackled by replaying events according to their timestamp order in a GLP coasting forward pahse, independently of the LP queue in which they are found. To this end, the traditional STF scheduler can be used during the GLP silent execution phase. This means that the actual silent reprocessing of the events is carried out by the same scheduling operation that is used in forward execution mode, although this time operating at GLP level. To prevent that duplicate messages are sent by LPs, the whole group is moved to the *silent execution* state. In this way, whenever some LP generates a new event destined to any LP in the system, this event is simply discarded—it was already sent before. To determine when the group should leave the silent execution state and enter again the

[8]Clearly, the actual checkpoint operation can be optimized using common literature techniques such as incremental checkpointing ones [18, 19, 25], which might fit larger grain LPs, just like GLPs. In our implementation, and in the presented experimental study, we rely on a baseline periodic non-incremental checkpointing scheme. A deeper study of the interactions between the GLP concept and optimized checkpoint/restore schemes, or even reverse computing schemes for state recoverability (see, e.g., [5, 7]), is planned as future work.

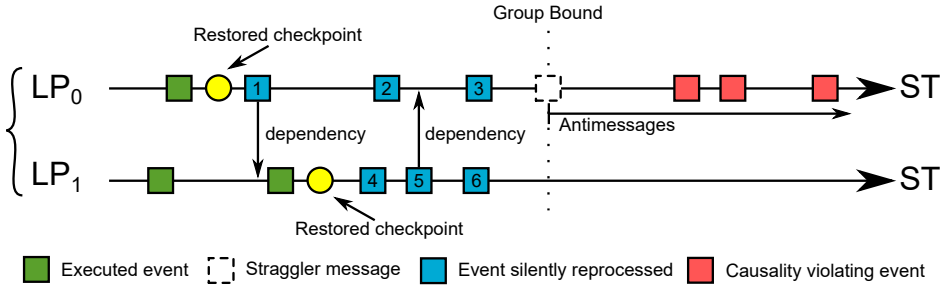

Figure 5: Inconsistent execution of traditional rollback/coasting forward phase with active groups.

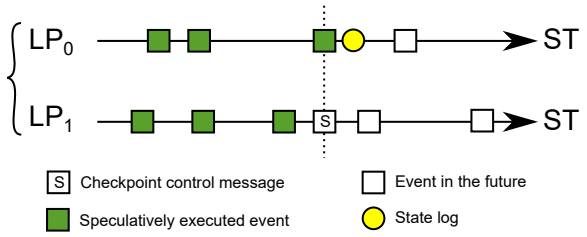

Figure 6: Group checkpoint control messages.

ready state, it is sufficient to check when the group bound is reached. At that point, all the events involved in the GLP coasting forward figure as already reprocessed. An example of this type of execution is depicted in Figure 7, where events e_1, e_2, e_3, e_4, and e_5 are processed in the correct order by the STF scheduler. Event e_3 is involved in a cross-state dependency, but it anyhow observes a consistent GLP snapshot.

In case a rollback operation falls before a GLP is revealed—which is detected by checking if the timestamp of a straggler message falls before the group revelation bound—the group is brought back to the wait-for-group state, so that cross-state dependencies are again runtime tracked according to the original proposal in [16].

3.4 Lifetime of GLPs

So far we have discussed our GLP proposal in terms of group determination, activation, and speculative execution. A question which should be still answered is related to the lifetime of a group: how long should a GLP exist in a given simulation run? While detecting when a synergistic execution starts could be easy, thanks to the runtime tracking of cross-state dependencies when LPs are not granulated, determining when such a phase ends is a non-trivial achievement, given the fact that when a group is revealed, cross-state dependencies are no longer traced (in fact, memory access to any state of the LPs belonging to the group is directly allowed by the runtime system to the worker thread that is in charge of managing the GLP). To this end, when a group is created, a simulation-time interval $\delta_{\mathcal{G}}$ is determined. This interval is computed as the average simulation-time increment that the model reaches in a given GVT phase. $\delta_{\mathcal{G}}$ is therefore a parameter which depends on the simulation model and its current execution phase, which captures the "speed" at which the parallel simulation is being carried out. As a first estimation, we set a GLP lifetime to $L_{\mathcal{G}} = k_{\mathcal{G}} \cdot \delta_{\mathcal{G}}$, where $k_{\mathcal{G}}$ is a per-GLP value which is initially set to 3.

Therefore, whenever a group is detected, we immediately schedule a *group-untie control message* to all the LPs in the GLP at simulation time $T_u = GVT + L_{\mathcal{G}}$, where GVT is the GVT value which was reduced when the group was detected. This event is associated with an additional counter in the group control block, which is initially set to the number of LPs in the GLP. Whenever such an event is executed by a LP (still at platform level, with no real dispatching of the application code), the counter is decremented, and when it reaches zero the group is considered as *untied*. LPs belonging to an *untied* group could be mapped to different worker threads when a LP rebinding phase is executed, and they again rely on the traditional Time Warp protocol, augmented with cross-state access tracking as in the original proposal in [16]. The group-untie control message should be kept in the input queue, as its effects are not transient: a rollback operation targeting a simulation-time instant which falls before the group-untie moment must re-activate the group, and therefore these control messages might be replayed.

This first estimation of $L_{\mathcal{G}}$ clearly needs to be improved at runtime. In fact, a synergistic execution phase which led to tying a number of LPs in a GLP could be either shorter or longer. To this end, whenever a group is determined, we generate a sort of *group fingerprint*, which is able to capture the size and the LPs which are belonging to the GLP. This fingerprint is based on a Bloom filter [2] using 3 hash functions, which is a space-efficient probabilistic data structure, telling whether an element is a member of a set. When a group is determined at turn i, we compute the Bloom filter B_i. At the next step, when groups are determined again, for each group we compute the Bloom filter B_{i+1}. Note that, by Algorithm 1, if the dependencies do not change significantly, the same LPs are tied into the same GLP. Therefore, we can compute the approximation of the intersection of the two Bloom filters according to the result in [20] as:

$$-\frac{m \ln\left[1-\frac{X_i}{m}\right]}{k} - \frac{m \ln\left[1-\frac{X_{i+1}}{m}\right]}{k} + \frac{m \ln\left[1-\frac{X_i|X_{i+1}}{m}\right]}{k} \quad (1)$$

where m is the size of the filters, k is the number of hash functions (3 in our case), X_i is the number of bits set in the i-th filter. $X_i|X_{i+1}$ denotes the bitwise OR between the two filters. Counting the number of bits set in a given word (a problem also know as the *population count*) could be a costly operation. There are nevertheless specific hardware supports for this (e.g., the `popcnt` instruction on modern x86 processors) or extremely optimized software routines for this [14].

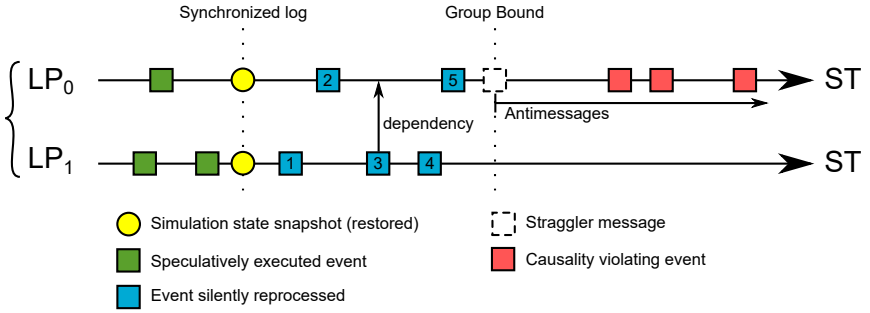

Figure 7: Group-rollback operation.

This number is an estimation of how much two instances of a GLP associated with a given ID are different[9]. We set the threshold τ_{dif}, which is used to apply a hill climbing algorithm to self-tune the value $k_{\mathcal{G}}$ of the group in the following way:

$$k'_{\mathcal{G}} = \begin{cases} k_{\mathcal{G}} \cdot (1+\alpha) & \text{if } dif_{\mathcal{G}} \geq \tau_{dif} \\ k_{\mathcal{G}} \cdot (1-\alpha) & \text{otherwise} \end{cases} \quad (2)$$

where $\alpha \in [0,1]$ is a tuning parameter which tells how fast the value $k_{\mathcal{G}}$ should be adapted, and $dif_{\mathcal{G}}$ is the outcome by Equation (1). On the basis of Equation (2), GLPs last longer in case they don't change very much, while they are untied more frequently in case the synergistic phases are short.

4. EXPERIMENTAL STUDY

We have implemented the support for GLPs within the ROOT-Sim [21] package[10], an open source Time Warp-based general purpose simulation platform hosting simulation models developed using the C programming language. This platform already offers the baseline support for cross-state synchronization as presented in [16].

In this section we provide experimental data achieved by testing our proposal running the implementation of a multi-robot exploration and mapping simulation model, as developed in [15] according to the results in [8]. In this model, a group of robots is set out into an unknown space, with the goal of fully exploring it, while acquiring data from sensors (e.g., cameras, lasers, ...) which are used to map the environment. The robots are equipped with enough processing power to elaborate the sensors data online (thus, the map is constructed during the exploration), so as to allow them to rely on the acquired knowledge to drive the exploration in a more efficient way. Specifically, whenever a robot has to make a decision about which direction should be taken to carry on the exploration, it is done by relying on the notion of *exploration frontier*. By keeping a representation of the explored world, the robot is able to detect which is the closest unexplored area which it can reach, computes the fastest way to reach it and continues the exploration.

The robots explore independently of each other until one coincidentally detects another robot. Whenever two robots enter a proximity region, they perform three different actions: i) they use their sensors to estimate their mutual physical position—recall that they are just in *proximity*; ii) they verify the goodness of their position hypothesis by creating a rendez-vous point (not to be confused with rendezvous control messages in the underlying Time Warp platform supporting granulation) in the explored part of the region, and trying to meet again there; iii) if the hypothesis is verified, they exchange the data acquired during the exploration, thus reducing the exploration time and allowing for a more accurate decision of the actions to be taken. Additionally, in case step ii) succeeds (i.e., the robots actually meet in the rendez-vous point), it means that the estimation of their respective position is correct. Therefore, they can form a *cluster*, i.e. they can start exploring the environment in a collaborative way. This collaborative exploration can take place in two different ways. On the one hand, they jointly define (by relying on *cost* and *utility* functions, as defined in [8]) their next exploration targets, so that they can minimize the time required for a complete environment exploration. On the other hand, they might decide to make a *guess* about the position of other robots (the total number of which is known) which are not part of the cluster yet. In the latter case, one of the robots (the one for which the utility/cost ratio is convenient) targets the hypothesized position. If a robot is found there, the aforementioned steps are carried out, so as to increase the knowledge of the environment.

Discovering the presence of a nearby robot is a crucial step while coding this simulation model. In fact, in case of reliance on classical PDES programming schemes not based on cross-state access, either the robots must communicate to each other their current position (thus exponentially increasing the number of exchanged messages, say cross-scheduled events, which in turn can limit the performance of the simulation), or they have to notify it to specific simulation objects (i.e., the regions), again increasing the number of messages exchanged. Additionally, to estimate the respective position of the robots, many simulation events could be required. In this specific case, these events should be marked with the same timestamp, thus requiring efficient (but non-negligible in cost) tie-breaking approaches, like the one in [13]. Third, exchanging map information could entail a data transfer non-negligible in size, posing a huge burden on the communication subsystem.

This model is therefore a good test-case for exploiting the innovative programming paradigm based on cross-state access, and to test the advantages from granulating LPs according to the new mechanisms we have presented (just supporting this programming model). In our implementation (as said aligned with the one in [15]), we rely on two different types of LPs, namely active ones (implementing the robots) and passive ones (implementing regions of the ex-

[9]Recall that the ID of a GLP corresponds to the ID of some LP belonging to the GLP.

[10]The full source code of our implementation is available at https://github.com/HPDCS/ROOT-Sim.

ploration environment). More specifically, the environment is represented as a square region, divided into hexagonal cells. This choice allows us to define a meaningful mobility model for the robots, and at the same time allows us to define proximity regions which are used by the robots to detect the presence of other ones in the nearby. Also, in our model, periodic events occurring into any cell are envisaged as the basis for modeling the evolution (inside the cell) of any phenomenon characterizing the dynamic change in the state of the explored region.

At simulation startup, each passive simulation object creates random obstacles (which prevent the robots from reaching any neighbour cell), mimicking a rescue scenario, where an open space is modified by an accident and the robots are used to explore it for rescue activities. At the same time, each passive LP instantiates in its private simulation state (by relying on a traditional `malloc` call) a *presence vector*. Each entry of the vector is associated with a specific robot. Whenever a robot enters a given cell, it explicitly informs the LP taking care of the cell's state by exchanging an event, piggy-backing a pointer to a buffer in the robot's simulation state which keeps the representation of the explored map. When the cell processes this event, it stores the pointer in the presence vector, which is then scanned to synchronize the information in the map. In particular, all the robots' states are in-place accessed, so as to copy the information from one state to the other. This operation clearly triggers cross-state synchronization and may lead to granulate LPs temporarily residing in a given area, together with the LP modeling the specific portion of the environment where they reside.

To test the GLP proposal we have compared the execution time for this simulation model when run with the granulation support, and without granulation thus running with the baseline cross-state synchronization protocol (labeled as CS). We have also run the same identical model on top of a serial engine based on a classical calendar-queue scheduler. Finally, for completeness of the analysis, we have run a version of the same model coded by only relying on the traditional paradigm where cross-state access is not employed/supported, thus basing the interactions among the different parts/entities in the model exclusively on the cross-scheduling of events across the different LPs. For all the tests we run a model with 1000 LPs, the 10% of which represent robots, and the remaining 90% represent sub-regions of the overall bi-dimensional region to be explored.

The hardware architecture used for running the experiments is a 64-bit NUMA machine, namely an HP ProLiant server, equipped with four 2GHz AMD Opteron 6128 processors and 64 GB of RAM. Each processor has 8 cores (for a total of 32 cores) that share a 12MB L3 cache (6 MB per each 4-cores set), and each core has a 512KB private L2 cache. For the parallel runs we configured the simulation platform to use 32 worker threads.

The total execution time for the simulations are reported in Figure 8 for the different settings of the underlying simulation engine (where each reported sample is averaged over 10 runs). For GLP-based runs, we have also considered the variation of the threshold parameter τ_{dep}, which we recall can be used to filter out cross-state dependencies that are less valuable (say, their volume is lower than others) while building the GLPs. Also, in GLP-based runs, the granulation process is actuated after the first GVT computation

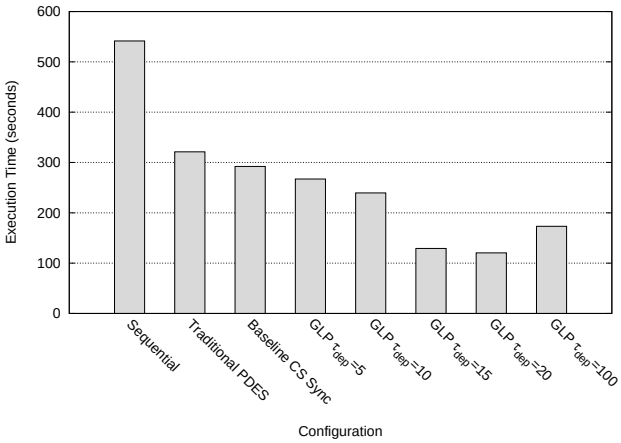

Figure 8: Total execution time.

round that is subsequent to the fading (due to lifetime expiration) of previously formed groups[11].

By the results we observe that all the platform configurations offering support for in-place cross-state access outperform the traditional PDES scenario, not admitting cross-state accesses. Nevertheless, the baseline CS synchronization scheme offers a limited gain, while the integration of the new granulation support leads to noticeable performance improvements. These improvements are better for larger values of the parameter τ_{dep}, indicating that a more accurate selection of actually valuable cross-state dependencies can drive significantly better granulation decisions, leading to more efficient synchronization dynamics, which lead the execution time to be almost 3 times lower compared to both the traditional PDES case and the baseline CS synchronization scheme. An additional observation concerns the step reduction of the execution time with GLPs when increasing τ_{dep} from 10 to 15. From the collected statistics we noted that this phenomenon is due to the fact that up to the value $\tau_{dep} = 10$, scenarios were generated where granulation involved multiple LPs representing distinct sub-regions, which does not favor concurrency since the model let each robot LP to move around, with the constraint of residing at any time in a single region. Hence, at any time instant, some event can only lead to cross-state access involving a region and its hosted robots. Different sub-regions, and their currently hosted robots, should be therefore allowed to execute concurrently to favor parallelism, by placing them in different GLPs. Nevertheless, our solution allows to be sufficiently resilient to this scenario. In fact, the first value of τ_{dep} where some thrashing phenomenon is observed (which in turn affects the overall performance) is observed only for a much higher value, namely $\tau_{dep} = 100$.

5. RELATED WORK

The LP granulation scheme we have presented can be seen as a means to bias synchronization dynamics in order to let a Time Warp system improve its performance in contexts where different portions of the simulation model exhibit more strict interdependencies (along different phases of the simulation run). The consequence would be an improve-

[11]In all the parallel runs the GVT period has been set to 1 second.

ment of fruitful usage of computing resources while carrying out speculative processing of DES models. This is the objective of classical load balanging/sharing approaches proposed in literature (see, e.g., [4, 10, 22]). However, these proposal are bound to scenarios where the interactions across the model portions are explicit and only occur via the classical event cross-scheduling approach (in-place state-wide access to the model state is not supported/considered).

More generally, to the best of our knowledge, there are not many proposals focusing on enabling non-explicit inter-dependencies between different LPs for the case of optimistic PDES. The work in [17] tackles the issue of transparently supporting the access to a shared state portion of the model by whichever event on multi-core Time Warp platforms, which is anyhow confined to global variables. This work presents state management/recoverability operations that allow concurrent event-handlers, CPU-scheduled to run events at multiple LPs, to observe a consistent global snapshot of such shared portion even in the case they do not notify their operations via explicit event exchange. Beyond a few technical differences—such as the reliance on application software instrumentation in [17] vs the employment of kernel-supported memory management facilities in order to runtime detect the materialization of an in-place access to some state portion in our proposal—the proposal in [17] does not entail mechanisms for dynamically determining the level of correlation of the memory (read/write) operations involving the overall model state. Rather, our granulation mechanism targets the dynamic discovery of such correlation so as to dynamically create islands of the model state where in-place access by the application code is enabled in a synchronization efficient manner.

In [6] the authors present the concept of Extended LP (Ex-LP). An Ex-LP is a collection of LPs that explicitly expose towards other (Ex-)LPs a certain amount of attributes, which can be implicitly accessed/modified. The access to exposed attributes is synchronized by relying on the Software Transactional Memory (STM) paradigm. In this sense, memory accesses to the shared portion of the state (the exposed attributes) is as well speculative, and could be subject to undo (via transaction aborts). This proposal anyhow requires pre-declaration of the state portions that can be subject to in-place concurrent access (since these must be accessed via STM demarcated code blocks) and there is no concept of dynamic clustering of different state portions in order to apply cluster oriented synchronization mechanism aimed at improving the runtime efficiency. Contrarily, in our proposal we do not require any pre-specification of what state portions will be accessed in-place by a same event-handler execution instance, and we support the dynamic clustering scheme with associated runtime change of the synchronization dynamics. Overall, we improve both flexibility of model coding (via the support for state-wide in-place access) and the potential for higher fruitful usage of resources.

The notion of *Kernel Process* (KP) is presented in [3]. A KP is a sort of PDES kernel-level thread instance that handles a collection of LPs by managing a single event-list of their events. The core objective is to optimize operations such as the fossil collection (memory recovery), which can pay-off even if false rollbacks can occur due to updates of the unique (merged) event list caused by stragglers or anti-messages (leading to rollback LPs in a KP that could still be considered as causally consistent). In our proposal the clustering of the LPs on a same thread is enabled dynamically (although with no explicit fusion of the event lists of the clustered LPs, rather by adopting a multi-list management approach), depending on the cross-state dependencies that are materialized at runtime and on the choice by the underlying granulation mechanisms. Further, our Time Warp environment supports a different programming model where in-place access to whatever state location is enabled. We feel that the event-list management features/optimizations at the core of KPs could be somehow integrated with our granulation support to further improve the performance of the runtime system.

The proposal in [1], still targeting multi-core architectures as we do, proposes a technique called Dynamic Local Time Window Estimates (DLTWE), in which each processor communicates time estimates of its next inter-processor event to its neighbors, which use the estimates as bounds for advancement. The proposal specifically targets spatial simulations, in which different (close) sub-volumes could be interested by a rollback operation. A selective rollback function is described, which allows to reduce the effects of rollbacks at LPs managing "close" entities. Contrarily, we do not impose any topology or predetermined relation across the LPs, which is an implicit outcome thanks to the different supported programming model (based on in-place state access everywhere). Moreover, we limit the effect of a rollback too for applications exploiting such a programming model by explicitly avoiding causal inconsistencies across LPs that are dynamically granulated together.

6. CONCLUSIONS

In this article we have introduced the concept of granular Time Warp simulation objects, and the design of a runtime PDES platform enabling the granulation process. This leads to dynamically clustering the baseline Time Warp objects, thus forming larger entities that advance in logical time as individual larger-grain objects, whose elements do never give rise to causality errors towards each other. This is achieved by jointly supporting a programming model where the simulation code can be written in such a way to access in-place the state of whichever object (either granular or not) while processing each individual event, an approach that stands as a valuable alternative to traditional PDES only based on event exchanges for coding interactions across the objects. Such cross-state accesses are what drive the formation of granulated objects, which are aimed at clustering the baseline objects that along specific execution phases shown larger volumes of cross-state dependencies. Also, this mechanism leads to runtime configurations where the level of parallelism is dynamically determined on the basis of the level of coupling of the objects, as determined by cross-state dependencies materialization. We tested our proposal against traditional Time Warp and a variant with cross-state support but no granulation, for the case of a multi-robot exploration simulation model run on a 32-core machine. By the study we report a 3x improvement in the model execution speed thanks to our proposal.

7. REFERENCES

[1] P. Bauer, J. Lindén, S. Engblom, and B. Jonsson. Efficient Inter-Process Synchronization for Parallel Discrete Event Simulation on Multicores. In *Proceedings of the 3rd ACM SIGSIM Conference on*

Principles of Advanced Discrete Simulation, SIGSIM-PADS, pages 183–194. ACM Press, 2015.

[2] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] C. D. Carothers, D. W. Bauer, and S. Pearce. ROSS: A High-performance, Low-memory, Modular Time Warp System. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

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

[5] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient Optimistic Parallel Simulations Using Reverse Computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, 1999.

[6] L.-l. Chen, Y.-s. Lu, Y.-P. Yao, S.-l. Peng, and L.-d. Wu. A Well-Balanced Time Warp System on Multi-Core Environments. In *Proceedings of the 25th Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 1–9. IEEE Computer Society, 2011.

[7] D. Cingolani, A. Pellegrini, and F. Quaglia. Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES. In *Proceedings of the 2015 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS, pages 211–222. ACM Press, 2015.

[8] D. Fox, J. Ko, K. Konolige, B. Limketkai, D. Schulz, and B. Stewart. Distributed Multirobot Exploration and Mapping. *Proceedings of the IEEE*, 94(7):1325–1339, 2006.

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

[10] D. W. Glazer and C. Tropper. On Process Migration and Load Balancing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):318–327, 1993.

[11] M. P. Herlihy. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

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

[13] H. Mehl. A Deterministic Tie-breaking Scheme for Sequential and Distributed Simulation. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, PADS. ACM Press, 1992.

[14] R. E. Odeh and D. E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms.*, volume 64. Addison-Wesley Professional, mar 1969.

[15] A. Pellegrini and F. Quaglia. Programmability and Performance of Parallel ECS-based Simulation of Multi-Agent Exploration Models. In *Proceedings of the 2nd Workshop on Parallel and Distributed Agent-Based Simulations*, PADABS, pages 395–406. LNCS, Springer-Verlag, 2014.

[16] A. Pellegrini and F. Quaglia. Transparent Multi-core Speculative Parallelization of DES Models with Event and Cross-state Dependencies. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS, pages 105–116. ACM Press, 2014.

[17] A. Pellegrini, R. Vitali, S. Peluso, and F. Quaglia. Transparent and Efficient Shared-State Management for Optimistic Simulations on Multi-core Machines. In *Proceedings of the 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 134–141. IEEE Computer Society, 2012.

[18] A. Pellegrini, R. Vitali, F. Quaglia, A. Pellegrini, and F. Quaglia. Autonomic State Management for Optimistic Simulation Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1560–1569, 2015.

[19] R. Rönngren, 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*, PADS, pages 70–77. IEEE Computer Society, 1996.

[20] S. J. Swamidass and P. Baldi. Mathematical Correction for Fingerprint Similarity Measures to Improve Chemical Retrieval. *Journal of Chemical Information and Modeling*, 47(3):952–964, 2007.

[21] The High Performance and Dependable Computing Systems Research Group (HPDCS). ROOT-Sim: The ROme OpTimistic Simulator. https://github.com/HPDCS/ROOT-Sim, 2012.

[22] R. Vitali, A. Pellegrini, and F. Quaglia. Load Sharing for Optimistic Parallel Simulations on Multi-core Machines. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):2, 2012.

[23] 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*, PADS, pages 211–220. IEEE Computer Society, 2012.

[24] J. Wang, D. Jagtap, N. B. Abu-Ghazaleh, and D. Ponomarev. Parallel Discrete Event Simulation for Multi-core Systems: Analysis and Optimization. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1574–1584, 2014.

[25] D. West and K. Panesar. Automatic Incremental State Saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, PADS, pages 78–85. IEEE Computer Society, 1996.