SAPIENZA
UNIVERSITÀ DI ROMA

Faculty of Computer Engineering
Computer Science and Statistics

Master of Science in Engineering
in Computer Science

Master Thesis

Runtime management of simulation objects' cross state dependencies in NUMA oriented parallel simulation platforms

Academic Advisor
Prof. Francesco Quaglia
Dr. Alessandro Pellegrini

Candidate
Francesco Nobilia

Academic Year 2014/2015

"Learn from yesterday, live for today, hope for tomorrow.

The important thing is not to stop questioning."

—Albert Einstein

ABSTRACT

Nowadays, the well acceptable solution for speeding-up and making very large and complex simulation models tractable is the parallel-DES (PDES) paradigm. In a speculative environment two simulation entities with different simulation times can reach the same portion of simulation state at the same wall-clock time. In this case the system has to manage their accesses always guaranteeing state coherence. For copying with this issue the entire model has been partitioned into distinct Logical Processes (LPs): each LP handles and models a portion of the whole simulated environment/phenomenon, and interacts with others by means of time-stamped event messages (local causality constraint) [1]. The reasons behind this disjunction are only technical. They impose a coding style that prevents the possibility that some LP can directly manage more than one state at a time. The purpose of this work is answering to the question raised by Fujimoto [1] whether building a shared state system by using messages only is the natural way to program simulation. Our solution is based over the concept that each LP can directly access the state of any other LP by means of synchronization phase. This behaviour is achieved setting-up each simulation object over a parallel memory view. Further, given that modern parallel machines are organized according to the Non-Uniform-Memory-Accesss (NUMA) model, we also provide approaches for making the access to memory slices associated with the parallel memory view efficient in NUMA

systems. We augment the ROme OpTimistic Simulator (ROOT-Sim)[2] with our proposal and we use this environment as test-bed. Finally, we demonstrate how our approach improves the simulation performance.

CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

Discrete Event Simulation (DES) is a type of simulation used to study systems where the changes in their states occur at discrete points in time. It is widely exploited for evaluating and analysing systems in many fields including computer and telecommunication systems, biological networks, military war gaming, online games, operational management and decision making. Timeliness in the delivery of simulation outputs is a relevant issue. The DES performance limitations have been overcome by Parallel Discrete Event Simulation (PDES). Instead of running a serial simulation, it exploits the natural parallelism available in, e.g., every modern multi-core processors. PDES is based on the idea of dividing the entire simulation model into distinct slices, namely Logical Processes (LPs) or Simulation OBJs (SOBJs), that are executed concurrently on the available CPU-cores. Thanks to partitioning and concurrent execution, PDES allows exploiting the computing power offered by parallel/distributed platforms in order to both speed-up model execution and make large models tractable. Any LP is implemented as a set of data structures plus a call-back function that processes simulation events by, e.g. accessing/updating these data structures [1].

The execution of the simulation model is a complex process that evolves according to a sequence of discrete events, namely instantaneous actions with an impulsive duration, managed by LPs. Executing events and following a

predetermined logical scheme allow LPs to pass through one state to another simulating the evolution of the target environment/phenomenon.

A PDES environment schedules LPs in non-decreasing time-stamp order by always dispatching the LP that handles the event with the minimum time-stamp. However, such a dispatching operations occurs in parallel on multiple CPU-cores, thus some synchronization mechanism is required to ensure correctness. There are two main approaches: the conservative one and the optimistic one. According to the former it is impossible that an event is executed out of time-stamp order along any LP. While the latter allows LPs to speculatively process their events under the optimistic assumption that the time-stamp order of events will never be violated. In other words, it leads to executing events without the assurance that no one will eventually generate an event in the logical past of some already processed event. If a violation arises, a rollback protocol is exploited to bring the LP back to the snapshot that stands right before the time-stamp of the event that caused the violation. The most common optimistic PDES protocol is the Time Warp [3].

The traditional interaction between LPs based over the cross-scheduling of events entails large computational and storage overheads both for handling messages and for storing them just in case of recovery procedure. Even though it is possible to build a shared-state system by using messages only, Fujimoto [1] raised the question of weather this is the natural way to program simulation applications.

Considering a battlefield simulation [4], a lot of information residing in each grid sector is required to be shared among many combat units, however in classical PDES we model each sector and each unit as an individual LP with its own private state. In the absence of shared variables, a message passing solution for reading and updating information in each sector should bee put in place, which might also lead the LPs handling sectors to become the bottlenecks

(e.g., due to flooding of event-messages for accessing their states). Also a pure shared-state approach may be not efficient because we would need to synchronize the LPs while accessing it so as to always return (and produce) values coherent with the LP's local virtual (logical) time. As Fujimoto affirmed, a more efficient approach would be the one of duplicating the shared information within the state of the LP that needs it, but a protocol would be required to ensure coherence among the various copies. He suggested to hide the coherence strategy in the underlying simulator. Starting from this idea dated 1990 some solutions have been designed to cope with the state-sharing issue, as a means for creating scenarios where LPs share (dynamically) the portions of the state that better fit the model semantic.

This work answers the Fujimoto question in the context of shared-memory multi-core architecture. We provide a more general programming and execution model than the traditional PDES. The state portion that can be access by each LP during its execution is not limited to its own state or to shared global variables only. Indeed, an LP is allowed to access the state of whichever simulation objects both in read and write mode . The idea is granting each simulation object the possibility of accessing the others LPs' states as in a sequential-style DES execution, where the latter pass to the former a pointer to their states inside the payload of simulation message. Nevertheless, targeting this situation in a sequential environment is trivial, while inside a parallel one it requires the creation of an advanced memory management architecture together with an advanced synchronisation mechanism that can guarantee consistency and progress. Furthermore, we cope with these issues in a transparent manner and targeting NUMA machines.

This solution has been implemented inside the ROme OpTimistic Simulator (ROOT-Sim)[2].

The rest of this thesis is organized as follow: CHAPTER 2 provides an

overview of theoretical aspects of simulation DES, CHAPTER 3 introduces issues related to memory usage and discusses relevant research literature results which will be taken into account during this work. CHAPTER 4 is dedicated to the ROOT-Sim engine and its subsystems relevant for this work. Our solution is detailed in CHAPTER 5, while CHAPTER 6 provides experimental results. Finally, CHAPTER 7 concludes the thesis with some considerations.

CHAPTER 2

Parallel Discrete Event Simulation

Discrete Event Simulation (DES) is a type of simulation used to study systems where the changes in their states occur at discrete points in time. It is widely exploited for evaluating and analysing systems in many fields including computer and telecommunication systems, biological networks, military war gaming, online games, operational management and decision making.
The first definition of *Parallel and Distributed Simulation* by Chandy and Misra [5] dates back to 1979 and the following definition of *Discrete Event Simulation* suffered of performance limitations. These limitations have been overcome by Fujimoto with the definition of *Parallel Discrete Event Simulation* (PDES) [1]: instead of running a serial simulation, it exploits the natural parallelism available in, e.g., every modern multi-core processors. Nowadays PDES is the well accepted solution for speeding-up the execution of simulation applications, and for making very large and complex simulation models tractable.

DES is based on the idea of executing a sequence of so called discrete events that modify the simulation state. A *discrete event* is an action such that the time of its beginning corresponds with its ending as well, and therefore it has an

impulsive duration. Based on the receiver's identity we have two different types of messages: a *local event*, if the sender and the receiver are the same simulation object, or a *remote event*, if the sender is different from the receiver.

PDES adds to previous definition the idea of dividing the entire simulation model into distinct slices, namely Logical Processes (LPs) or Simulation OBJects (SOBJs), that are executed concurrently on the available CPU-cores. Each LP handles and models a portion of the whole simulated environment/phenomenon, and interacts with others by means of time-stamped event messages, namely discrete event temporally related by *Local Virtual Time* (LVT) . More in detail, according to the classic PDES idea an LP can only handle its own state and the interactions across the LPs take place by message passing, say cross-scheduling of events. Therefore simulation is the result of cooperation between LPs by means of non deterministic sequences of events that lead to the computation of a global predicate. Non deterministic means that executing twice the same simulation could generate the same outcome passing through different intermediate steps.

Figure 2.1 shows the typical architecture of a simulation platform, namely the *Simulation Kernel*. It has a multi-layer organization where on top we can find LPs that are handled by the underling simulation kernel instance. Kernel instances are divided according to the multi-thread paradigm: as soon as they are activated they take control of all available CPU cores inside the hosted machine precluding the operating system chances of moving them around, in this way we can improve both the performance and locality. More than one machine may take part in the simulation and therefore libraries for supporting communication across different machine are required (e.g MPI library [6]).

Exploiting the parallelism has introduced new problems in addition to the one formerly presented in DES paradigm. They are related to the fact that we are now targeting asynchronous environments where events happen at irregular

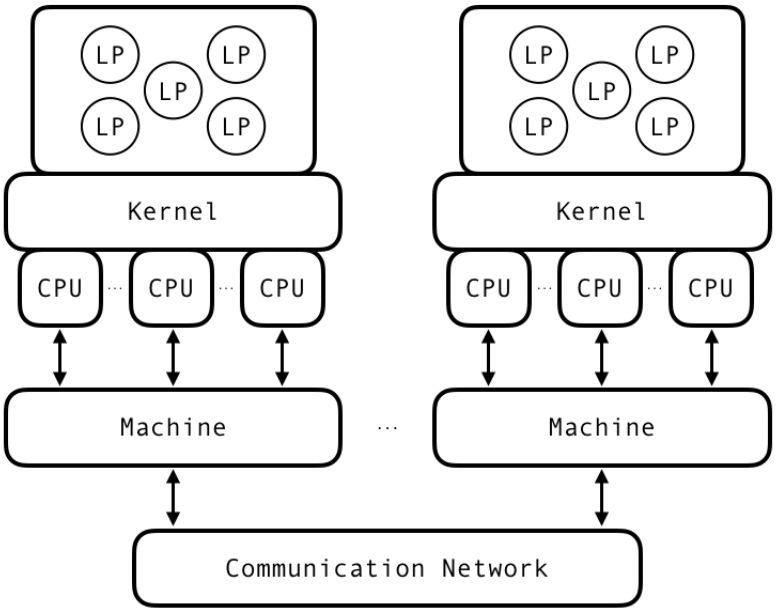

Figure 2.1: System Architecture: Symmetric Multi-Threaded Simulation Kernel

intervals. In the past years these problems have been widely studied and thousands of different solutions have been provided. The most important issues are:

1. *state consistency*: the global state of simulation model must be always coherent with environment/phenomenon that we are simulating and with the sequence of generated events: inside an asynchronous system it is highly possible to receive a message with time-stamp smaller than the receiver's LVT and therefore counteractions must be designed for guaranteeing coherence
2. *scheduling*: LPs are dispatched in non-decreasing time-stamp order by always dispatching the LP that handles the event with the minimum time-stamp, hence the scheduler enhances evolution of simulated environment generating the sequence of changes that will modify the state trying to select them in an order that does not violate causality relations
3. *global clock*: in a multi-core environment and even more in a distributed system there is the lack of a shared global clock. This type of instrument

is fundamental for driving the evolution of simulation without violating causality and therefore the determination of a *Global Virtual Time* (GVT) is required

4. *message exchange*: any message received by any LP is stored inside a FIFO (first-in-first-out) queue sorted by time-stamp for preserving their sequence and causality. Due to the system asynchronicity there is no guarantee that a straggler message[1] will be never generated

Summarizing: the management of event queues, communication, synchronisation and execution orders are operations demanded to the simulation kernel.
Due to the intrinsic asynchronicity of these systems, it is highly possible that an unsorted message queue, that saves messages in the same order in which they were arrived, stores an event e before e' with e being at time-stamp t and e' at time-stamp t' such that $t' < t$. Executing e before e' is inconsistent. e' will find a state already modified by e at time t while its action is related to t' and therefore the execution of e' will lead the simulation into an inconsistent state. This is the problem of coherence over the observability of the simulation state. A possible solution is to schedule LPs in non-decreasing time-stamp order by always dispatching the LP that handles the event with the minimum time-stamp. However, such a dispatching operation occurs in parallel on multiple CPU-cores and so it may be not enough: simulation object LP_x may send an event at time t_x to simulation object LP_y while LP_y is at time t_y with $t_y > t_x$ (see Figure 2.2). In this case the simulation kernel has to apply a technique for restoring a state of LP_y coherent with the time-stamp t_x to guarantee that from now on all following events will be executed respecting the correct order as long as a new straggler arrives.

[1] A straggler is a message with time-stamp smaller than receiver's LVT

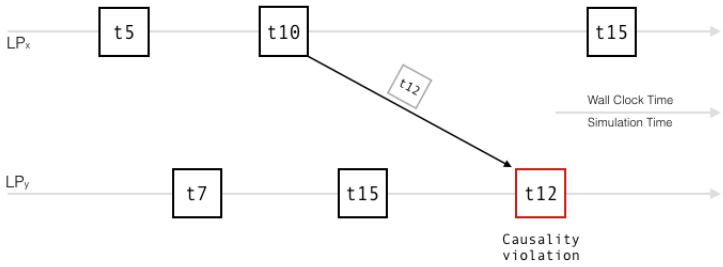

Figure 2.2: Example of causality violation

2.1 Synchronisation Strategies

Scheduling LPs in non-decreasing time-stamp order does not guarantee correctness: dispatching the LP that handles the event with the minimum time-stamp is not sufficient because this operation occurs in parallel on multiple CPU-cores. For copying with this issue, some synchronization mechanisms had been introduced. Particularly, there are two synchronisation approaches: the conservative and the optimistic one. According to the former it is impossible that an event is executed out of time-stamp order along any LP. On the other hand the latter allows LPs to speculatively process their events under the optimistic assumption that the time-stamp order of events will never be violated. In other words, it leads to executing events without the assurance that no one will eventually generate an event in the logical past of some already processed event. If a violation arises, a rollback protocol is exploited to bring the LP back to the snapshot that stands right before the timestamp of the event that caused the violation. There is also a third solution: the hybrid approach that simply mixes the other two approaches trying to exploit their strength points [7, 8, 9, 10].

2.1.1 Conservative Synchronisation

Conservative synchronisation was the first approach for targeting PDES environment. It tries to a-priori avoid causality errors: an event is executed only

if it is recognized as *safe* [5, 11, 12, 13, 14, 10]. This synchronisation scheme distinguishes two types of events:

- *safe* events: an event e at time-stamp t is declared safe if during the dispatching of e inside the system any other event has time-stamp bigger than t and the simulation kernel can guarantee that from now on an event with a time-stamp smaller than t will never be generated
- *unsafe* event: if at least one of the constrains described above does not hold then event is declared unsafe

In this way any event execution will never generate causality violations. This solution is implemented by establishing a FIFO channel without loss for each LP. At each run the channel will deliver to the corresponding LP the event with the minimum time-stamp already presented inside the channel, in other words events are executed in non-decreasing time-stamp order. In case of unsafe execution the corresponding LP waits for the late messages, as soon as they will arrive the LP can restart its execution. Obviously, this behaviour can lead the system into a deadlock: all LPs are waiting for same messages that will never arrive since all simulation objects are blocked. For avoiding this scenario some solutions have been provided [5]. For completely avoiding deadlock scenarios between LP_x and LP_y, the system has to guarantee that time-stamp order of messages sending by LP_x to LP_y is non-decreasing. If so, the last event received by LP_y and sent by LP_x is the lower bound of all next feasible time-stamps that they will use for exchanging messages from now on. If LP_x is waiting for LP_y while the latter is waiting for the former, system generates an artificial message, called *null message*, sent by LP_x to LP_y, with time-stamp equal to t meaning that LP_x will never generate new messages with time-stamp smaller than t. If deadlock could not be avoided, there is a mechanism able to bring the system back to a snapshot in which deadlock no longer exists.

The definition of safe event is strictly related to the concept of *lookahead*: if a simulation model can certainly affirm that all events starting from virtual time t until event at $t + L$ are safe, it means its lookahead is equal to L. This value

is bigger than or equal to zero since in the worst case it represents the time interval between two consecutive events. Setting an optimal value of lookahead is not trivial since it depends from the model that we are simulating : if it is too small it can hamper performance since a lot of already arrived events are considered as unsafe. Some solutions exist for improving this limit: they are based on the idea that we can pre-process a small sequence of unsafe events and check, after their completion, if the simulation model still holds a coherent state[2].

Conservative Synchronisation has three main advantages:

- *aggressiveless*: each simulation step leads the model in a coherent state, according to the synchronisation scheme there is no event execution that produces an incoherent state
- *riskless*: each computation of either an intermediate or a global predicate is always correct
- *minimum synchronisation between simulation objects*: each computational step always advances the logical time: it is impossible to roll back the LVT of at least one simulation object. Therefore computing a global predicate is always possible without entailing any kind of synchronisation.

On the other hand, the exaggerated conservativeness produces a simulation that does not perform and advances slower: the parallelism generates only few advantages because it is not entirely exploited. Let us consider two simultaneous events e_1 and e_2, if they have a causality relation, say $e_1 \rightarrow e_2$, the system will execute e_1 and then e_2, if instead they are not related they could be executed in whatever order but the system still forces the previous execution order event although it is useless and reduces performance.

[2]This solution reminds optimistic synchronisation scheme, but they are actuated in a different manner [15]

2.1.2 Optimistic Synchronisation

In contrast with the conservative philosophy, the optimistic synchronisation still executes events in a non-decreasing time-stamp order, but it dispatches the LP that handles the event with the minimum time-stamp without checking if it violates local causality constrains thus being oriented to higher parallelism. In this way optimistic means advancing the simulation as far as possible toward the future without checking whether any internal or external event has been safely generated according to LVT. The lack of this safety led to the designing of techniques able to bring the simulation objects back to a snapshot that stands right before the time-stamp of the event that caused the violation.
The most common optimistic PDES protocol is Time Warp [3]. It dispatches the LP that handles the event with the minimum time-stamp, meanwhile it checks if this execution is consistent or not. If a causality error is later revealed, meaning that a simulation object has received a straggler event, the simulation flow is stopped and a coherent snapshot is reloaded into the system. A *coasting-forward* procedure is actuated for correctly taking into account the new information stated by the straggler. All events with time-stamp bigger than straggler are annihilated and then the *rollback* phase is triggered: all events from the snapshot to the straggler are re-executed accordingly the actual correct order (see Figure 2.3).

The work in [3] suggests the adoption of th *Global Virtual Time* (GVT) that determines the lower bound of any future rollback. It is computed by a distributed algorithm across all LPs [16, 17]. This lower bound identifies a logical time before all events which are considered as *commited*: no rollback can occur at any time smaller than GVT, meaning than no straggler with a time-stamp smaller than the GVT will be ever generated.
As counter part, high parallelism entails additional overhead due to memory consumption for taking snapshots and computational power loss due to rollback.

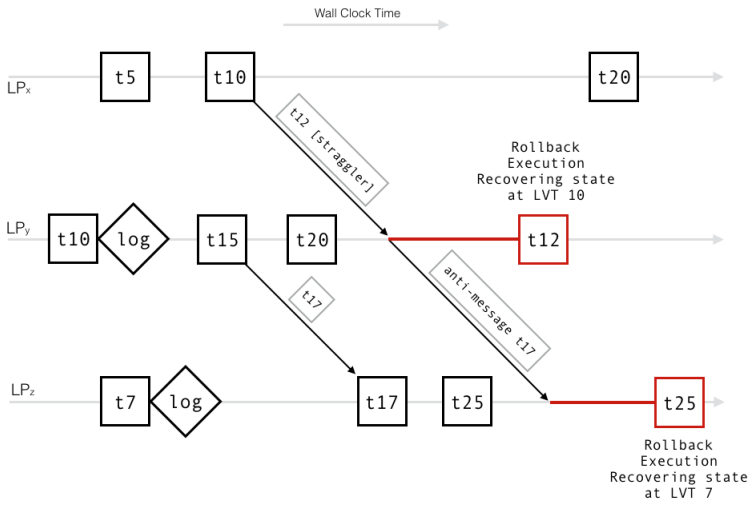

Figure 2.3: Rollback work flow

State Saving Policies

Before introducing the rollback operation, let us explain how the snapshots restored by rollback are computed.

In this work [18] introduced the *reverse computation* technique: instead of restoring a previous snapshot, they suggests to invert the execution flow of the application code for reproducing the required state, this can be done in an automatic or semi-automatic way. However, techniques for *State Saving* (SS) are believed to be more progressive [3, 19, 20]. The SS challenge is to have a solution transparent to the application level programmer that takes care of which check-point police is better to use, how long should be the period between two consecutive checkpoints, reducing at minimum the required resources and understanding when it is possible to delete old snapshots.

As follow a briefly preview of the most used technique for taking snapshots.

Formerly introduced in [3], *Copy State Saving* (CSS) is the easiest way for dealing with SS. At the end of each event execution, the simulation kernel takes

a complete snapshot of the entire simulation object state and its required data-structures useful for rolling back. Each snapshot is marked with the time-stamp of the last-executed event, therefore in case of a straggler the snapshot that stands right before the straggler's time-stamp is simply restored and nothing has to be reprocessed for reaching the required LVT. Of course, this solution implicates a huge overhead in terms of memory consumption. To avoid the execution of all available memory, snapshots that are no longer useful for future rollback executions must be deleted. This operation is called *fossil collection* [19]: all snapshots with a time-stamp smaller than the GVT can be safety deleted thanks to the GVT definition itself. Actually, fossil collection takes care of freeing message queues from messages with time-stamp smaller than GVT as well.

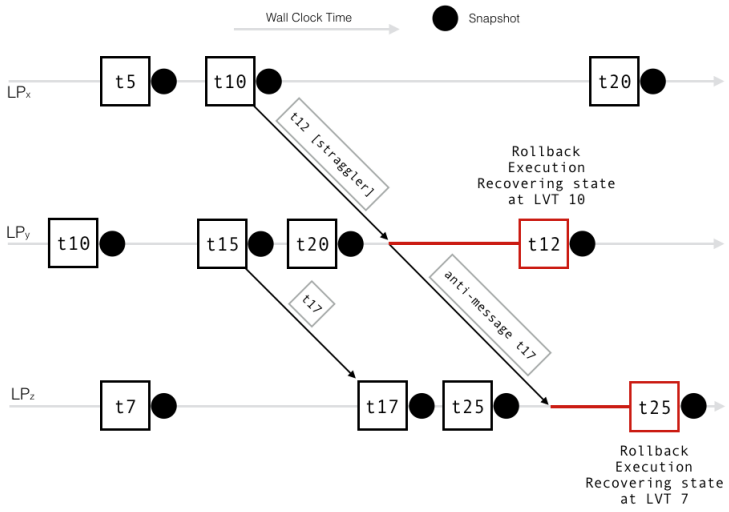

Figure 2.4: Example of rollback in case of CSS

To reduce the overhead of CSS, *Sparse State Saving* (SSS) solutions have been provided. Instead of taking snapshots between each simulation event, they try to log states sparsely according to either a constant period (*Periodic State Saving*-PSS) or a variable one (*Adaptive State Saving*-ASS). Initially designed in [21] and then improved in [22], PSS or *State Skipping* takes snapshots every n messages where n is a parameter that can be set before launching the sim-

ulation. Now, when a simulation object receives a straggler a snapshot with exactly the required time-stamp may not exist. In this case, the system selects the snapshot which stands right before the required time-stamp and then re-executes in *silent mode* all events that exist between snapshot's logical time and the straggler one. Re-executing events due to rollback is named *coasting forward* or *state rebuilding*, while silent mode means reprocessing events without sending messages to other LPs since they had already been sent during the original execution. Each reprocessed event must return the same result that produced during the first execution otherwise rebuilding the correct state will be impossible; in other words each event's execution must be deterministic, only message arrival order is non-deterministic. This behaviour is called *piece-wise determinism* (PWD) [23]. PSS optimises memory consumption by reducing the computational power: the coasting forward phase has a relevant cost. Hence it is crucial to set an optimal value for the check point period: a too small value entails an inefficient memory usage while a too large one reduces performance. ASS solution has exactly this purpose: according to the model it tries to identify the best value of checkpoint period [24, 25, 26, 20].

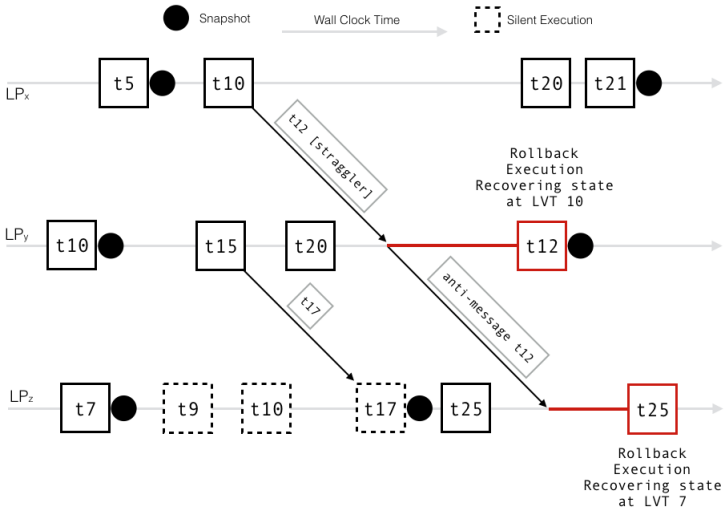

Figure 2.5: Example of rollback in case of PSS

In scenarios where the states of simulation objects are too big SSS solution

is too expensive in terms of memory consumption. The only possible alternative is the *Incremental State Saving* (ISS) approach. Instead of logging the entire state, the underling system saves only the memory areas modified after the last checkpoint. Of course it produces a smaller overhead in terms of memory consumption and computation power loss for taking snapshots.

Rollback Policies

The introduction of Time Warp paradigm has led to the design of mechanisms able to a-posteriori detect the materialisation of causality violations due to straggler events. This mechanism, called *rollback*, restores a consistent simulation state and then restarts the execution taking into account also the straggler event (see Figure 2.5). Rolling back the state and re-executing some events are not enough, as we can see in Figure 2.5, the LP that is rolling back has to retract all messages that it has sent in the logical time interval between instant in which it has received the straggler and straggler's time-stamp itself. This undoing is actuated by means of *anti-messages* that notify positive messages' receiver that the involved messages has should be annihilated. When an LP receives an anti-message, if it is temporally successive to its LVT, it simply deletes the corresponding message from its input queue, on the other hand if the anti-message is temporally before its LVT , it has to roll back into a consistent state. This is the *State Save & Restore*. Its counter part is the *Reverse Computation*. Instead of sending anti-messages, it sends a *negative message* that applies the same operations of the corresponding positive message but with opposite result. The same behaviour is actuated in case of straggler message: the receiver has to apply all required negative messages in order to restore a consistent state coherent with the simulation time of straggler. The main difference within these two approaches is that the latter generates rollback longer than the former: according to the State Save & Restore is sufficient to restore the coherent state for bringing the simulation object back to an older simulation time, on the other

hand Reverse Computation has to apply all negative-messages from th simulation time in which the straggler has been received until the straggler's time. As said before, the most used technique is State Save & Restore.

2.1.3 Hybrid Synchronisation

In simulations where many zero-lookahead events occur, the Time Warp protocol performance is reduced: the more zero-lookahead events happen, the higher is the rollback frequency. Hence in [27] hybrid solution between Conservative and Optimistic synchronisation has been discussed. This solution fits models where the number of LPs exceeds thousands. Starting with a Conservative approach, as soon as the scheduler encounters an unsafe event the simulation will continue via optimistic synchronisation. There are two possible approaches: either events are executed optimistically just inside a slice of simulation trajectory and therefore rollbacks are bounded inside the involved trajectory (*Limited aggressiveness*), or the traditional optimistic approach is performed but then the entire model may be subjected to rollback (*Unlimited aggressiveness*). In the second case a *temporal window* is required for limiting the duration of speculative execution. This hybrid solution is called *Local Time Warp* (LWT).

2.2 State Management

For targeting PDES environments the entire simulation model has to be divided in disjoint slices called LPs or simulation objects where each one of this substates model a portion of the entire physical phenomenon. How this LP is composed from the coding point of view has not yet been discussed.

Each simulation objects is implemented as a set of data structures, namely *state variables*, plus a call-back function that processes simulation events by, e.g. accessing/updating these data structures. Each variable depicts a physical

feature of the real environment that the LP is representing. Any change in the physical environment is represented as a variable update [28]. Therefore a simulation program defines a collection of state variables that evolves across the simulation logical time following the logic depicted by itself. We can summarize the code of simulation program as follows

Algorithm 1: simulator

```
1: procedure SIMULATOR
2:     INIT( )
3:     while ¬EMPTY_QUEUE( ) do
4:         mex ← RM_STF_EVENT( )
5:         EXECUTE(mex)
6:         SEND_NEW_MESSAGES( )
7:     end while
8: end procedure
```

Sending messages can be seen like a request/query of LP_x toward LP_y for updating the state of LP_y (*pull event*) or for retrieving the required information for modifying the state variable of LP_x (*query event*). Concurrently dispatching different objects over all available CPU cores leads to concurrent updates of state variables. If LP_x and LP_y share a state variable while they are at different simulation times, any update of LP_x over the wall-clock time will leave the shared variable in an inconsistent state for LP_y and vice versa. The only way to prevent this type of causality error is imposing two limitations: the so called *local causality constraint* [28]

- sharing state variables in PDES environment is not allowed
- LP_x can trigger updates over the state of LP_y only by message passing

The whole simulation state must be divided into *state vectors* (SV), in the correspondence of one SV per LP. This assumption guarantees that a parallel simulation where each SOBJ agrees with local causality constraint will generate exactly the same result of serial simulation if events are executed in time-stamp order and in case of simultaneous events they are executed in the same order in both environments. Of course, this constraint is sufficient but not necessary

for assuring that no causality error will occur: a straggler message may still be generated and two simultaneous events causality unrelated may be executed by two different SOBJs according to a different order without generating consistency issues. Furthermore it entails nothing about correctness since correctness is in charge of synchronisation protocols (see Section 2.1).

So far we have considered only state variables that are initialised during the initialisation procedure of simulation, without considering the possibility that the allocation of new state variable may be needed during the simulation. This new requirement adds a new challenge for our synchronisation strategies: the rollback procedure has to take into account any kind of simulation object's memory shape and therefore if we want to instantiate new memory buffers during the simulation run we need a mechanism to notify rollback about the new allocation. Informing it only upon memory creation is not enough, we must communicate also each memory free operation. Without countermeasures, a run-time allocation of memory may be rolled back and consequently a memory leak may be generated since the rollback procedure cannot free it. On the other hand, if the system rolls back a memory free operation the involved area beforehand owned by LP_x may have been reallocated for LP_y and therefore both of them will unconsciously use the same memory area, a situation that generates an unpredictable behaviour of model. The solution presented in [29] exactly fits these issues establishing an intermediate level between the simulation kernel and the underling operating system by wrapping any memory related operation. The possibility of dynamically allocating memory during simulation gives the possibility of building state variables with sizes that evolves over the simulation time.

Fujimoto [28] suggests to see the snapshots of state variable as a special messages that LPs send to themselves containing information regarding their state variables. The value of the time-stamp related to the message/snapshot

is the time-stamp of last event that has updated variables. In this way, in case of rollback the system has just to scan the per LP queue that stores snapshots and selects the state with the required simulation time.

The implementation discussed above is completely orthogonal with the concept of *global variable* described in [30]. They provide support for managing global variables satisfying transparency, non-blocking and dynamic constrains, and including a rollback protocol for PDES environments.

CHAPTER 3

Memory Issue

The cost of reaching high performance exploiting high level of parallelism and speculative computation lies in huge memory usage. For coping with causality errors a PDES environment has to periodically take state snapshots of each simulation object; in this way it will have the required information for dealing with rollback operations. In simulation models where the number of involved simulation object exceed thousands (such as agent-based demographic simulations [31]) or in situation where the simulation states are extremely large (such as distributed cloud data store models [32]) the amount of memory occupied by each snapshot is striking as well as the memory occupied by the state that is now depicting each simulation object. In this context where uncommitted information can exceed the capacity of RAM, memory trashing and the resulting system degradation must be avoided as well a mechanism for improving data locality is required[33].

The amount of memory used during the simulation is not the only issue regarding memory. According to the classic PDES idea each simulation object can only access the memory areas bounded within its own state while any access outside these limits is not allowed for causality issues. The interactions across simulation objects take place only by message passing. Even though it is possible to build a shared-state system by using messages only, Fujimoto [1]

raised the question of whether this is the natural way to program simulation applications. Due to speculative execution it is highly possible that two simulation objects with a different LVT reach the same area of simulation state at the same wall-clock time. In this case the system has to manage their accesses in such a way that after their actions the model will still hold a coherent state. Traditionally, for copying with this issue the entire simulation state S has been partitioned into per-LP substates S_i, with the possibility that each LP can only access its own state and it can interact with other LPs only by message passing (local causality constraint). This can be summarized with the following *state/ interaction constraint* in a system with N LPs:

$$S = \bigcup_{i=0}^{N-1} S_i \quad \wedge \quad S_i \cap S_j = \emptyset \quad \forall i \neq j \tag{3.1}$$

Equation (3.1) constraints dictates restrictions that are caused by only technical limitations. For handling rollback operation and the relative forward execution the traditional solution expects that simulation state is divided into smaller portions, in this way in case of rollback only the involved simulation objects have to rewind their states while all other can carry on the simulation. Those technical limits generate clear disadvantages to application level programmer. Indeed, simulation models such as agent-based demographic simulations [31] or distributed cloud data store models [32] could improperly use the cross-scheduling of events: the former has to exchange thousands of events just for communicating the value of a single variable, while the latter generates events hosting huge copies of state sections.

3.1 Memory Usage vs NUMA

In [33] a memory management architecture able to attenuate latency for accessing memory area in case of execution over multi-core *Non-Uniform-Memory-*

Access (NUMA) platform is described. The challenge of NUMA environment is related to the fact that the amount of available RAM and the number of available CPU-cores always increase, therefore actuating uniform memory access is not trivial. The idea behind the NUMA memory architecture is to have memory areas with different access latencies according to their location: each processor can rapidly access its local memory, whereas it gets slower when addressing shared memory or, worse still, of others processors. In the context of multi-thread Time Warp platforms running on top of NUMA, they create an advanced memory manager able to

- store the data of each single simulation object into private memory: each SOBJ is instanced over a set of memory pages that are disjoint from the memory of other SOBJs
- guarantee that all memory pages of a SOBJ are handled by NUMA node that hosts the worker thread which dispatches it, in this way low latency and high throughput are ensured when the worker thread accesses the SOBJ's information

Linking simulation object's data and NUMA node is done in both static and dynamic way. In the latter case a daemon thread periodically checks whether a simulation object has been migrated to a different NUMA node and therefore memory migration is required. This solution is completely transparent to the application level programmer that can still exploit libraries for dynamically allocating the state of simulation objects.

3.2 State Sharing

Clustering simulation objects using a non-blocking and fully transparent approach, allowing high speculation by means of optimistic synchronization, and a dynamic scheme well fitting the simulation model semantic is a topic still

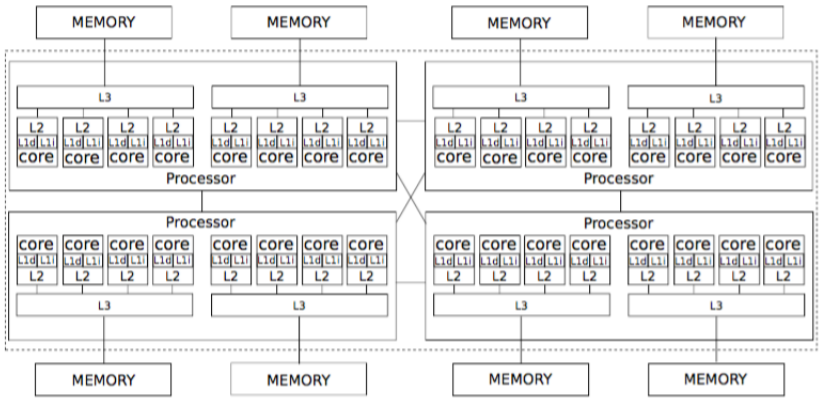

Figure 3.1: NUMA Architecture

under investigation.

The idea is to build a system that allows LP_x during its execution to access the memory slices keeping the state image of LP_y .

This problem was originally studied in [34] and then in [35]. During the past years some solutions have been developed for optimising and making the formerly proposed approaches coherent with modern hardware and operating systems.

We can divide the solutions in two different types: (i) the ones that want to share a partial view of the LPs' states and store this information inside shared variables, and (ii) the ones that aim to share the entire state image.

In [36] sharing is in charge of an *extra-LP* that handles all data that other LPs want to share, thus working as a server, for which a new rollback protocol must be adopted. This particular LP employs a multi-version list for storing its variables. Whenever another LP wants to modify the shared state, it communicates to the server the new value by a write operation and then the *extra-LP* will add the new data within the multi-version list. During a read operation, the

server selects the most appropriate data version according to local virtual time of the requesting LP (say the client). If during a write the local simulation time of the writer is smaller than the one of the value stored at the tail of the list, a causality violation materialises hence a rollback is required. Both read and write operations are actuated by message passing primitives. This solution satisfies none of the previous listed capabilities. From a performance point of view, the centralised node may be a bottleneck: it processes all requests sequentially limiting the evolution of simulation, while we are looking for something that can solve the problem through a non-blocking (say concurrent) way. Everything is left to the programmer himself: he has to call the proper API's for interacting with the shared data, as in the case of the (M,N) Atomic Register.

In [37] the concept of *state query* is introduced. Any LP that wants to access another LP's state can send a message to the corresponding owner (query event). Then it will wait until the handler sends back the required information (read event). As before, a causality violation may occur if queries are processed optimistically. In this case the corresponding anti-event is entered in the system. Again, there is no transparency and no wait-free behaviour. This approach resembles the one suggested in [1].

The idea of implementing shared variables by *global* is presented in [38]. This choice allows the developers to design read and write operations in-place, however they always need locks to ensure mutual executions. Also, the programmer must specify which LPs will use the shared variable.

The *middleware* presented in [39] and [40] implements shared variables but, as the word middleware suggests, it has been designed for a distributed environment and moreover without a speculative synchronization protocol. It exploits a request/reply protocol based on timestamp-ordering.

The idea of threads that want to concurrently access shared information in atomic way is strictly related with the concept of Transactional Memory. These

can be either hardware or software, which are also called *Software Transactional Memory* and [41] shows how it is possible to solve our problem by using them. However, this solution shows two drawbacks: application-level developers have to explicitly mark transactions, and there is no way to support rollback of committed transactions even though this is a fundamental tool inside optimistic environments.

The solution in [30] provides support for managing global variables satisfying transparency, non-blocking and dynamic constrains, and includes a rollback protocol for PDES environments. This proposal has been designed for ANSI-C based simulation models and has been integrated and made available as free software within the ROme OpTimistic Simulator (Root-Sim)[2]. It exploits the capabilities of a free software instrumentation tool called Hijacker [42] plus the Compare&Swap (CAS) hardware support to build *non-blocking version lists* where all operations (read/write) are done in-place since every data and metadata are stored within the same address space that is accessible by the concurrent threads that execute the LPs. The lock-free property is guaranteed by the CAS instruction and each list item stores the timestamp related to the event that has updated the value in order to avoid the ABA problem. Everything is done in a fully transparent manner. At compile time inside the assembly code, Hijacker finds each *mov* operations, understands if it is either read or write and then substitutes it with a proper injected code that allows the simulation to deal with global variables. As in [36] and [37] a casualty violation may arise but just in case of writes, and therefore [2] has been augmented with necessary functions for correctly rolling back wrong read operations upon writes occurring out of timestamp order. Read operations never rise violations because the old versions are always kept until memory pruning. However, this solution does not provide the possibility to share the entire state across all the LPs.

With the intention of targeting the optimistic synchronization protocol over multi-cores machines, [43] is the first solution in which LPs can directly access

the states of other LPs. The authors define the concept of *Extended LPs* (Ex-LPs) that are sets of LPs: a LP can access all states of its-mates without synchronization. The scheduling protocol guarantees that for each set of LPs just one element per group is executing at any time. This solution has two problems: no transparency is provided because specific APIs must be called in order to access the states; and the group composition must be known in advance, the programmer groups all LPs that he believes having close causal relationships.

In [44] and [7] the concept of *Space-Time Memory* (STM) presented in [45] has been used to address sharing states in PDES environments. Instead of considering memory as a linear array of values accessible by a single spatial "coordinate", Chandy and Sherman suggest to see it as a two-dimensional array where together with the idea of space there is also time dimension. The purpose is to share states among LPs. They partition the two-dimensional spaces into distinct regions, each one assigned to a single process that is in charge of computing the values of state variables stored there. The computation ends as soon as a fixed point is reached. Ghostand and Fujimoto suggest the creation of this shared object and the implementation of such behaviour achieving simpler coding and better performances enabling also rollback operations. Their abstraction contains three types of operations for creating, reading and modifying an object that must be directly called by programmers. Read and write operations cannot be invoked more then once over the same object by the same event because these return a handle to a "memory page" that stores the value, hence all future actions will be actuated through it. Both operations exploit two locks: one is used to know which event that has the lowest timestamp is now writing the object and one guarantees mutual execution. The used data structure is a linked-list sorted by timestamp, therefore the aforementioned handle is merely a pointer to the required list item. In order to modify the data structure, the list must be fully scanned adding a huge overhead.

3.2.1 Static vs Dynamic and Partial vs Full Sharing

The aforementioned solutions can be divided in two main categories: static and dynamic. With the former we indicate all alternatives that allow LPs to sharing their state only before the simulation is started and do not provide any facility to tune the share scheme according to simulation model evolution. Instead, the latter includes all those alternatives that augment the former ones with the possibility of run-time state sharing.

In both categories there are solutions that allow to share the LPs states completely or partially. Partial share relies on global variables [38, 30], or on a sort of server LP [36], or on Software Transactional Memory [41]. There is a sort of API that allows LPs to store some information inside a shared area, and the interaction with this API is left to the application-level programmer with no transparency whatsoever with respect to memory direct access. However, a centralized entity, like in [36], will likely become the bottleneck of the entire architecture. Also the absence of dynamicity in [38, 41] is a significant limitation: in both cases we need to know in advance which are the LPs that will use shared variables. Achieving transparency with additional compile-phases like in [30] has been shown to be viable limited to global variables (not the heap). The work in [37, 43, 44, 7, 45] describe how to share the entire state. [37] uses a message passing mechanism for allowing LPs to query others in order to access their state, entailing however huge overhead (indirect access). [43] defines how to access directly any state. The first one divides LPs in subsets in which everyone can access the states of any other, but these groups must be predetermined before the simulation is started. The approaches in [45, 44, 7] allow a complete access to the entire state.

However, none of the aforementioned solutions provide a mechanism leading

to dynamic sharing of LPs' states in a non-blocking and fully transparent manner, and still providing speculation. However, based on these solutions we claim that the possible approaches to target the hit are two: either we can rely on multi-version linked list like in [36, 30, 44, 7, 45], or we can augment the PDES environment so as to directly access the actual state like [43]. In both cases the read/write operations should be executed in-place like in [38]. The tradeoff is granting direct access to the states paying the cost of locks since the hit LPs must be blocked while the reading/writing one is actually working, or taking advantages of version-lists where causality errors are materialised only during write operations paying huge memory consumption. Furthermore similarity to how [43, 44, 7] have suggested, each group must be handled by a single thread: there is no reason to divide the LPs of a single group over two or more threads, it will cause only additional overhead due to synchronisation and information sharing.

3.3 Considerations

Today, PDES synchronization paradigms able to allow simulation objects to share their whole state in a completely transparent way towards the application programmer are still lacking. Our aim is to provide this synchronization scheme as well as allow the programmers to code in a sequential style. We provide a more general programming and execution model than the classic PDES. The state portion that can be accessed by each simulation object during its execution is not limited to its own state or to the shared global variables only. Indeed, each LP is allowed to access the state of whichever simulation objects both in read and write mode. The idea is to grant each LP the possibility to access the other LPs state as in a sequential-style DES programming model, where the latter pass to the former a pointer to their states inside the payload of a simulation message. Nevertheless, targeting this situation in a sequential environment is trivial, while

inside a parallel one it requires the creation of an advanced memory management architecture together with an advanced synchronisation mechanism that can guarantee consistency and progress. Furthermore, we cope with these issues in a transparent manner.

Our solution is completely orthogonal to the possibility to migrate memory inside a NUMA environment. Our aim is to provide each simulation with a set of logically contiguous, private and pre-reserved memory pages that it can share with other LPs on demand. After migrating memory from one NUMA node to another, the memory pages that were previously logically contiguous are still contiguous: each NUMA node manages all available memory as a single address space.

CHAPTER 4

Symmetric Multi-Threaded Optimistic Simulator

In this chapter we describe our reference architecture as well as our test-bed platform: ROme OpTimistic Simulator (ROOT-Sim) [2].

4.1 The Reference System Architecture

Out target architecture is called Symmetric Multi-Threaded optimistic simulation kernel for massively multi-core architectures with NUMA facilities (see Figure 2.1). It is pyramidal organised: on top there are LPs, underneath we find the simulation kernel instance that handles LPs, further there are CPU-Cores over which simulation kernels are spread. At the bottom of the architecture we find a communication layer: since multiple machine should take place in the simulation we need a mechanism to put them in contact (message passing-based communication network based on MPI library [6]). All involved machines follow the same system architecture. With the intention of achieving high performance exploiting the intrinsic parallelism of architecture and favouring memory locality, simulation kernels are divided according to a multi-threaded programming

paradigm: during the start-up phase each worker thread is statically assigned to an available CPU core thus blocking any attempt of underlining operating system,Äôs scheduler to spread worker threads around. Each LP has two timestamp-ordered queues able to guarantee high-performance: input queue and out queue. The former contains all received events, while the latter stores all sent events and it is useful for sending anti-messages during rollback operations. Since executed events are not eliminated, the input queue is both a future-event list and a past-event list, there is a pointer that keeps track of last-correctly executed event, say the *bound* event. LPs are dispatched following the "Smallest timestamp first": the event with smaller timestamp is always selected before the other that holds a next event with bigger timestamp. As Time Warp [3] suggests, rollbacks are executed only when they are really necessary: as long as the Select-Timestamp-First scheduler does not select an LP that holds a rollback state, the real rollback is not executed (*lazy rollbackl*). In this way the frequency of rollback is significantly reduced.

To take the most out of Symmetric Multi-Threaded simulation kernel a synchronisation mechanism is required for guarantee the correctness of simulation. It must take into account two main points:

- running in kernel mode without explicit synchronisation mechanism for avoiding scalability problem and improving performances
- avoiding loss of locality for completely exploiting the benefits of caches

Executing according to data partitioning paradigms satisfies the former condition, while the latter is naturally fit from multi-threading paradigm: all worker threads of a simulation kernel execute within the same address space and therefore all data structures related to whatever LP are accessible by any worker thread.

These two conditions have driven the design of top/bottom-half mechanism for those actions that lead a simulation object to potentially cross the boundaries

of another simulation object while the former accesses the data-structures of the latter. Whenever the former has to execute such a task, instead of blocking the latter, it updates the top-half data-structure without triggering an immediately finalisation. As soon as the latter has completed the tasks stored inside the bottom-half data-structures, it checks whether some new tasks are available, and if so, it will finalise them. This behaviour resembles the Linux task queue. For guaranteeing safety during concurrent accesses to top/bottom-half data-structure, a spin-lock array with as many entries as the number of simulation objects is owned by each simulation kernel, in other words a mutual-exclusion paradigm is provided. Techniques have been applied for ensuring high performances while using spin-locks. A typical application for the aforementioned data-structure is when LP_x has to access the bottom-half queue of LP_y due to new message/antimessage destined to LP_y while they are handled by two different worker threads. A similar situation occurs when LP_y has to handle the message reception.

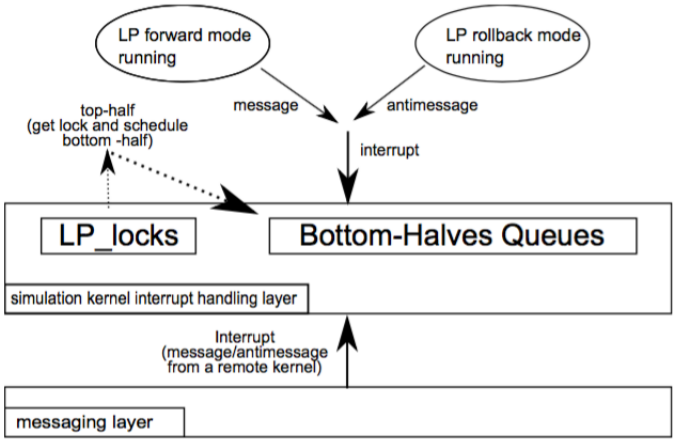

Figure 4.1: Top/bottom Halves Architecture

For coping with locality, we establish a wall-clock-time window during which a matching between LPs and worker threads is created. The affinity mechanism entails that a worker thread of a specific simulation-kernel instance is not allowed to run every LP hosted by that kernel. Instead, it handles just a subset of

them. For each LP that worker thread has in charge, it executes the following operation:

1. flushing the bottom-half queues of all simulation object controlled by each LP
2. selecting the LP that has to execute according to a time interleaved mode

Note that affinity between worker threads and LPs can change over time taking into account the variations of the amount of available worker threads. Due to the division a per thread data-structures, called $dispatched\_info[]$, are created. They take track of what LPs are handled by the current worker thread. It is just an array of pointer to the original shared state data-structure. In this way during the execution of *Smallest Time-stamp First* scheduling algorithm, each worker thread accesses only the private data-structure without interfering with others. Furthermore, using a shared data-structure would have caused lower performance due to continuous cache invalidation: the update operations would anyway invalidate the cached portion of the scheduler state on all worker threads, or would require cache fill, hampering performance. In this way each worker thread accesses only the LP's state pointed by its local $dispatched\_info[]$.

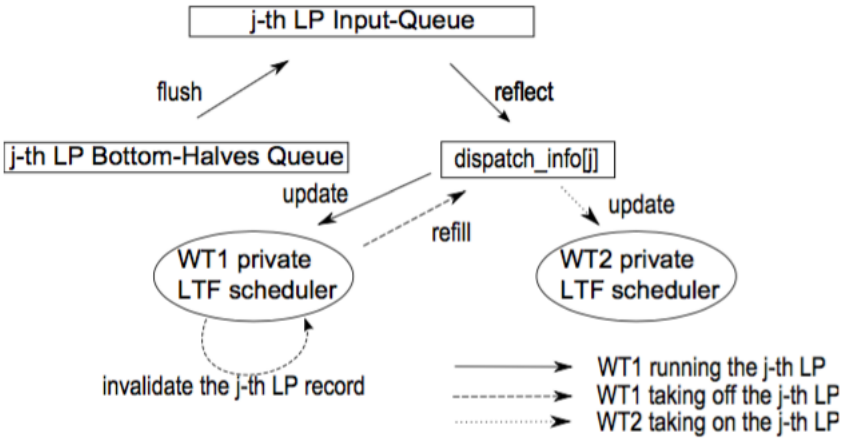

Figure 4.2: LTF Scheduler Management Upon Normal and LP Handoff Phases

With the purpose of maximising the global event rate, the affinity between worker threads and LPs can be modified also taking into account the amount

of work-load computed by each LP. The new division can be periodically re-computed during the GVT determination. The power reallocation algorithm computes a *Knapsacks* algorithm, that uses as weight parameter the work-load. For each LP_x hosted by simulation-kernel instance k_i, with $i \in [1, K]$ $x \in [1, numLP^{k_i}]$ where $numLP^{k_i}$ is the cardinality of k_i, the workload factor L_x is computed by k_i. Taking into account the simulation events currently registered in the input-queue and that will be executed in the next run, the esteemed distance in the future equal to the last GVT advancement that new events execution will cause weighted by the average CPU time for event processing, the equation to compute the work-load is:

$$L_x = \frac{q_x \times \delta_x}{LVT_x^{q_x} - LVT_x^1}$$

where q_l is the amount of pending events within the event-queue of LP_x with timestamps that fall within the interval of interest, LVT_x^i is the timestamp associated with the i-th pending event along the queue, and δ_x is the average CPU requirement for event processing by LP_x along that chain of pending events.

In massively multi-core architectures and even more in a distributed system there is lack of a shared global clock. This type of instrument is fundamental for driving the evolution of simulation without violating causality and therefore the determination of a *Global Virtual Time* (GVT) is required. According to the traditional definition of Time Warp, the GVT value determines the lower bound of any future rollback, in other words it is the global minimum across all worker-threads of messages/anti-messages timestamps that are either into the message-queue or that have already been incorporated into the event-queue. This computation is divided in two phases:

1. each worker thread computes the *local minimum*: the minimum timestamp of events kept by the event-queue
2. the minimum of all local minimum is computed by a distributed and wait-free algorithm where all worker threads participate.

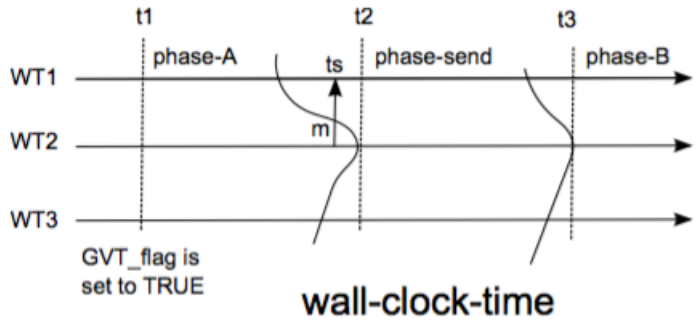

Figure 4.3: GVT Computation Phases Example

In our architecture we assume that the PDES system runs according to a dual-mode scheme where we distinguish between *application* vs *platform* modes. A worker thread enters in application mode as soon as it dispatches a simulation object, while it will renter in platform mode after the ending of each simulation event. In this context we consider the live state of a simulation object as recoverability data. Furthermore, any memory allocation/deallocation operation is not demanded to the standard `malloc` library: they are transparently intercepted by the underlying PDES environment and redirected to proper allocators.

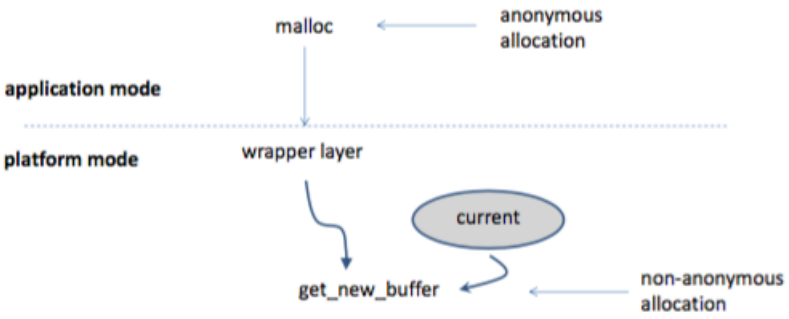

Figure 4.4: The Dual-mode Execution model

Upon calling the standard-library memory management API, the underlying environment identifies which is the invoking simulation object and then performs a non-anonymous memory allocation/deallocation operation. `get_new_buffer(int sobj_id, size_t size)` function allocates memory knowing the identity of the

caller simulation object. It exploits the underling open-source DyMeLoR allocator [29]. DyMeLor pre-allocates large memory segments by the original malloc library, and then it partitions these big segments into chunks. It uses compact bitmaps for tracking who is the user of each chunk and exhibits facilities to provide memory recoverability. DyMeLoR is also able to handle unrecoverable memory, in this case it is exploited by platform level code that hence does not require recoverable facilities.

Since we are targeting a NUMA environment, we need a NUMA allocator able to pre-reserve memory segments that can be delivered to the overlying (user-level) chunk allocator in non-anonymous way. `void* allocate_segment(int sobj, size_t size)` fits exactly this scenario. In case of NUMA environment, the user-level allocator calls the NUMA allocator for obtaining the required segment instead of calling the traditional malloc. As in the previous case, the NUMA allocator pre-allocates memory on-demand returning it to the invoker.

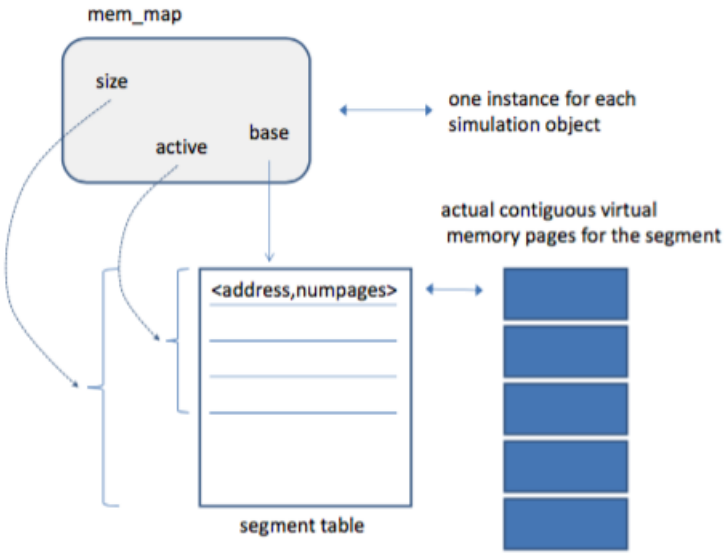

Figure 4.5: `mem_map` Data-structures

Pre-reserving is supported via the POSIX `mmap` system-call, which allows for

validating in the process memory map a set of contiguous virtual pages, whose global size complies with the size of the segment allocation request.

As Figure 4.6 shows, in case a worker thread has to allocate a buffer for platform level usage, say a unrecoverable memory area, the `get_new_buffer` function of the DyMeLoR user-level allocator for the unrecoverable case is called. This returns a pointer to a non-anonymous segment pre-reserved by DyMeLoR where the buffer resides. Instead, if the caller requires recoverable memory, the recoverable version of `get_new_buffer` service is called.

Thanks to the above architectural organization, the NUMA memory manager guarantees that the set of virtual memory pages destined to keep event buffers, live state and recoverability data for any individual simulation object is actually disjointed from the set of virtual memory pages used for storing data associated with other simulation objects hosted by the multi-thread PDES system.

Let us now discuss how these virtual memory pages are allocated on the different NUMA nodes. `set_mempolicy` system-call is called during the initialisation of our NUMA manager for ensuring that empty-zero memory is materialized on the NUMA node associated with the CPU-core where memory access is performed. This strict binding, together with the idea that memory pages keeping chunks of a specific simulation object are accessed only by the worker thread to which the object is bound, guarantees that physical memory allocation takes place exactly on the NUMA node associated with the CPU-core where the thread is running. It is clear that this architecture fits only situation where worker threads are statically bound to a specific CPU-core (such as when running with the `sched_setaffinity` service posted) and the simulation objects are statically bound to a specific worker thread.

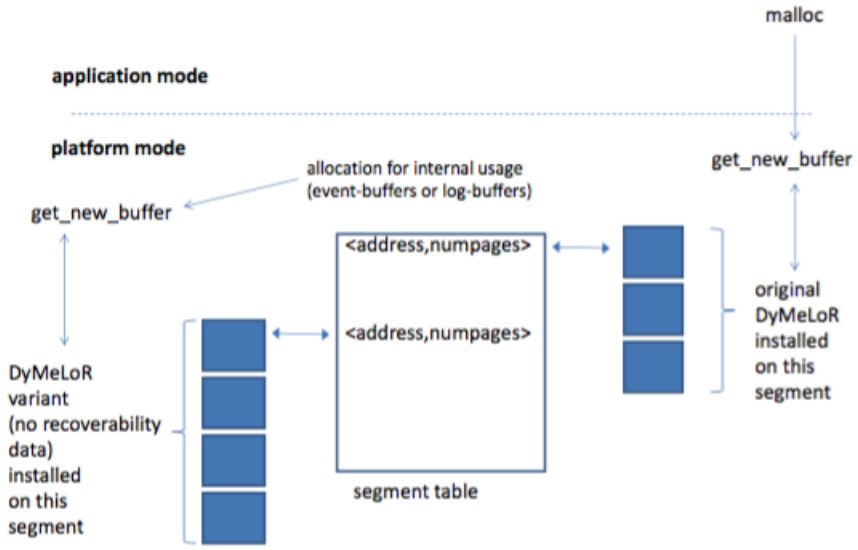

Figure 4.6: Memory Allocation for Platform and Application Usage

To cope with dynamic scenarios a page migration daemon, called `pagemigd`, is added to the previous architecture. It runs a set of CPU non-intrusive threads that periodically move memory pages associated with the memory map of a specific simulation object to a target NUMA node. Worker threads can trigger a move request via `void move_sobj(int sobj_id, unsigned target_node)`. In this case `pagemigd` accesses the data structure associated with the object to move, and migrates segments registered within the segment table of that simulation object.

The move is actuated via `move_pages` Linux system call, that takes as input the virtual addresses of pages to be moved and NUMA nodes toward which the move has to be actuated (see Figure 4.7).

As said before, `pagemoved` can be made up of multiple threads, each one in charge of checking migration requests, and migrating subsets of the simulation objects. Having multiple threads augments the probability that a move request is promptly executed. Furthermore, the performance is less hampered by partitioning the whole migration across multiple threads. Note that due to `pagemigd`

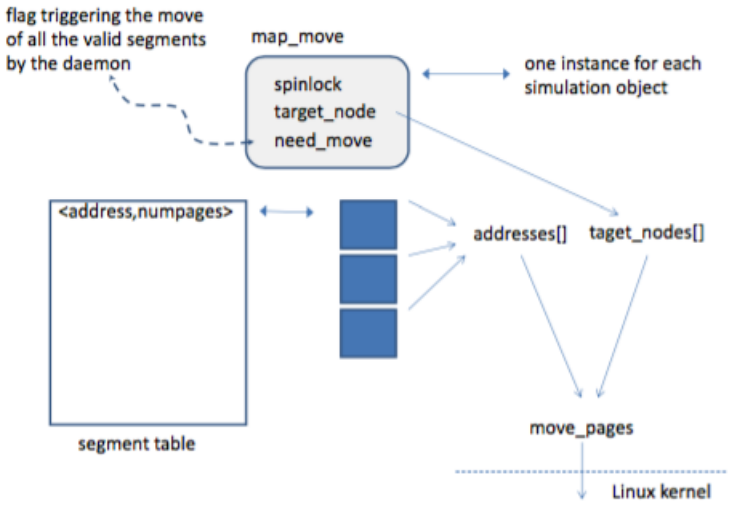

Figure 4.7: Segment Migration Operation

daemon no additional rollbacks are likely generated.

4.2 Simulation Engine

In this Section we discuss the technical organization of ROme OpTimistic Simulator (ROOT-Sim)[2], a open source PDES simulation kernel relying on the Symmetric Multi-Threaded Optimistic synchronization paradigm. It comes as a static library which can be linked to executables implementing simulation models using the ANSI-C programming standard [46], as if they were completely sequential.

In particular, the user can organize the code in as many functions/files as needed, can perform any I/O operation during the simulation (keeping in mind that I/O operations can degrade performance) can use dynamically-allocated memory to build the simulation state. No regular entry point is required for the application-level code, as entry points for the application code are specified by

ad-hoc APIs.

The actual simulation is based on events: each LP processes events, and its advancement in the LVT is connected to their execution. LPs communicate via messages or via global variables. As the Time Warp protocol suggests, *ROOT-Sim* forces a logical identity between events and messages. Each message contains an event that must be scheduled to some destination LP. Each event is identified by a numerical code, which is defined by the application-level logic and therefore more than one type of message can be used. The application-level programmer has to implements a `struct` for each event type, where the content of a message (an event) must be specified. Furthermore, there is an unique identifier number associated with each message that must be specified whenever a message is sent.

Each LP has its own execution context and its own stack. They are both implemented as user-level threads (ULT). Each stack lives in the LP's stock, so we enforce a complete separation between the simulation-kernel-level and the LP-level data structures. Context switches are executed relying on POSIX `setjmp`/`longjmp` API functions. Thus, activating an LP also means changing the execution context from the worker thread to the LP.

Each logical process has its data structures enclosed in a `LP_struct` data structure (resembling what the Linux kernel does for processes), thus enforcing modularity end easiness to extension. All execution context information is maintained into this structure.

Concurrent execution is based on the notion of worker thread. At simulation start-up, *ROOT-Sim* as many threads as the number of available CPU-cores and

spreads them one per core, in this way it can exploit all computing resources available in the underlying architecture. The *ROOT-Sim* divides LPs in disjoint subsets and assigns a subset to each worker thread. During the simulation, this binding can be periodically modified taking into account the work-load of each simulation object in order to maximize the overall performance by adopting a load sharing policy.

The first event, namely the `INIT` message, executed by each LP is automatically sent by *ROOT-Sim*. The application code related to this event must be always specified because it initialises the simulation state of related LP by a sequence of `malloc()` calls. At the end of `INIT` the other events can be scheduled allowing the simulation to start. The state created during the `INIT` handling is considered as the initial part of the LP's simulation state, and will pass via a pointer to the API that executes any event that can be overridden by application programmer. In this way LPs' states can arbitrarily grow/shrink during the simulation's execution, just relying on additional `malloc()`/`free()` calls.

The general architecture of the current version of *ROOT-Sim* is shown in Figure 4.8.

4.2.1 Supported APIs

The interaction between application-level code and the simulation kernel is established via three core API that must be necessarily implemented in the simulation model to be compliant with the library. Then, the rest of the code can be implemented in any way, albeit respecting the ANSI-C standard. These APIs are:

- `void ProcessEvent(int me, time_type now, int event_type, void *event_content, void *state)` is the callback that supports the actual processing of simulation events, and it is used by the kernel to give control

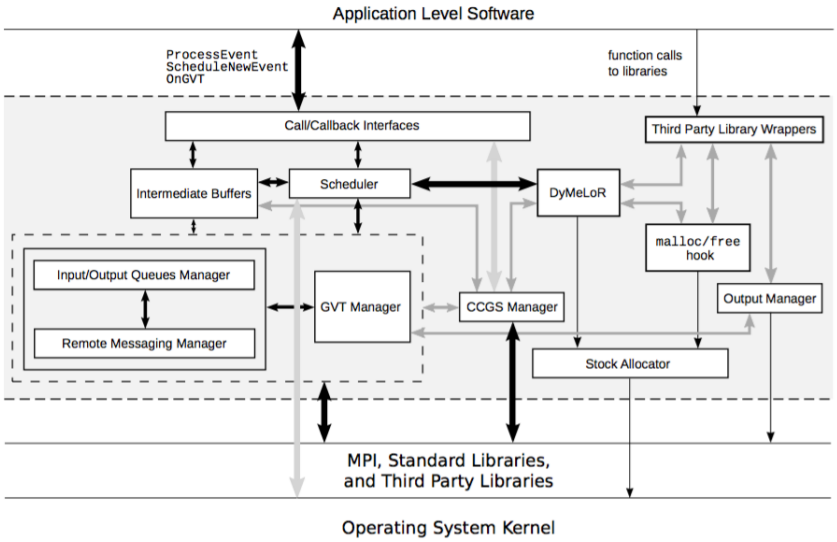

Figure 4.8: Root-Sim Architecture

to the application layer. Its parameters are:

- `me`: the ID of the LP being scheduled
- `now`: the current value for the local clock
- `event_type`: the numerical code for the event to be processed
- `event_content`: the information regarding the event that `me` is going to execute
- `state`: the current LP's state.

Inside of `ProcessEvent()` the execution is fully speculative: the events that are executed might be eventually undone in a way that is completely transparent to the application-level programmer. However, if non-rollbackable actions are executed inside this function they cannot be undone, for instance if the programmer prints some texts on the screen during the execution of an event that will be eventually rolled back, the output generated will not be reverted.

- `void ScheduleNewEvent(int receiver, time_type timestamp, int event_type, void *event_content, int event_size)` is a function that allows injecting a new simulation event within the system, to be destined to whichever

simulation object. Its parameters are:

  - `receiver`: the ID of the destination LP
  - `timestamp` the LVT associated with the event to be processed
  - `event_type`: the numerical code for the event to be processed
  - `event_content`: the information regarding the event that current LP is sending
  - `event_size`: the event memory size

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

- `bool OnGVT(void *snapshot, int gid)` is a callback that gives control to the application layer by also providing a committed snapshot of the simulation object. Its parameters are:

  - `snapshot`: the newer committed state related to the current LP
  - `gid`: the ID of the current LP

  The execution of `OnGVT()` is therefore not speculative. This means that, e.g., any I/O operation within this function is perfectly safe, and therefore it can be used to gather statistics on the ongoing simulation. We note that, since the timestamp associated with `snapshot` refers to the committed portion of the computation, it is forbidden to call `ScheduleNewEvent()` within `OnGVT()`, because this might induce a rollback operation of already committed events. `OnGVT()` additionally implements a distributed termination control: since `snapshot` is a portion S_i of the Committed and Consistent Global State (CCGS) S, a global predicate can be locally eval-

uated on S_i. If the model determines that the simulation is completed for that particular LP, `OnGVT()` can return the `true` value. *ROOT-Sim* will collect all return values, and in case all the LPs agree, the simulation will stop.

Other important facilities are:

- `void SetState(void *new_state)` that allows the LP to manually specify which is its simulation state.
- *numerical library* which has functions that generate random numbers according to several distributions: `Random()`, `Expent()` (exponential), `Normal()`, `Gamma()`, `Poisson()`, and `Zipf()` distributions. This library is noteworthy: the same sequence of numbers will be deterministically produced if a rollback operation is performed. It maintains one seed per each LP, pseudo-randomly drawn by an initial master seed which can be either randomly computed at simulation start-up, or manually specified by the user. This gives full control on the model's execution, giving the possibility to re-study the same configuration (determined by the initial master seed) which will give the same final outcome, independently of the actual events' execution pattern.
- `int FindReceiver(int topology)`: returns an LP's neighbour according to some geometric topology uniformly at random. Valid values for `topology` are `RING`, `BIDRING`, `LINEAR`, `HEXAGON`, `SQUARE`, `STAR`, `MESH`, which describe respectively a ring, a bidirectional ring, a linear vector, a square region divided in several hexagonal cells, a square region divided in several square cells, a star topology, and a mesh.

4.2.2 Internals and Subsystems Organization

The main simulation loop of *ROOT-Sim* is shown in Algorithm 4.2.2. It is concurrently executed by each worker thread

```
procedure Simulation-Loop
    while End == false do
        Receive remote messages
        Process Bottom Halves
        if GVT interval has passed then
            start GVT computation
        end if
        e_next ← events associated with timestamp T_min among the bound LPs
        if  thene_next is a straggler
            rollback LP in charge of processing e_next to T_e_next
        else
            switch context to LP in charge of processing e_next
        end if
        if CCGS tells simulation is complete then
            End ← true
        end if
        process outgoing messages
        if GVT computation complete then
            execute Fossil Collection
        end if
    end while
end procedure
```

4.3 A Code Example

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

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

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

so we rely on a `switch` construct to demultiplex them:

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

The code logic is fairly simple: upon `INIT` event, the LP's state is `malloc`'d and initialized, and an initial packet is sent to the LP itself. Whenever a `PACKET` event is received, a local counter is increased, and a packet is sent back to a random LP in the simulation environment. Timestamps are computed according to an exponential distribution, exploiting the internal `Expent()` function. `OnGVT()` is the second callback to be implemented, which performs a local check on the LP's state. If the number of packets passed through the LP is smaller than `PACKETS`, then the simulation cannot be halted yet:

```
40 bool OnGVT(lp_state_t *snapshot, int gid) {
41         if (snapshot->packet_count < PACKETS)
42                 return false;
43         return true;
44 }
```

CHAPTER 5

Cross-Accessing Logical Processes' States

According to the traditional definition of Parallel Discrete Event Simulation (PDES) [1], the simulation model is divided into distinct simulation objects that will be concurrently dispatched. It has led to the definition of simulation objects' states that are completely disjointed and to event executions that can access only the memory areas bounded within the executer's state. Due to this limitation, the execution of whatever event cannot directly access any valid memory location outside the boundaries of the running LP state: developing simulation code according to sequential style is no longer possible. If during its execution, LP_x has to partially or totally access also the memory view of LP_y, the programmer has to a priori map this behaviour over explicit message exchange. This mapping should take place during the code design/development phase. This concept promotes parallelism but limits the performance adding huge overhead to the entire system. Our aim is to relax this memory restriction favouring speed-up.

Nowadays, thanks to the spread of shared-memory parallel machines, such as multi-core and SMP machines, there is the possibility to directly share the

state information between different entities by means of shared memory and/or unique address space. Today, the synchronization paradigm providing such behaviour is still lacking inside simulation platforms. Our aim is to provide this synchronization scheme as well as allowing the programmers to code in a sequential style.

The idea is granting each simulation object the possibility to dynamically and directly access the local state of any other involved objects as in a sequential-style DES programming, where the latter passes to the former a memory pointer. Such pointer is the same one that the LP uses to dynamically allocate its memory during the simulation and therefore it references exactly the original state information. The LP that will receive this pointer will be able to access the local state of the sender both in read and write mode. This requires the creation of an advanced memory management architecture together with an advanced synchronisation mechanism that can guarantee consistency and progress. Furthermore, we cope with these issues in a transparent manner. Henceforth, the programmer will be able to treat simulation objects' states that are no longer disjoint. During the execution of whatever event, any valid memory location stating a portion of the state of whichever object can be accessed. From now on, we will refer to this new kind of memory dependency as *cross-state* dependency, which is complementary with respect to the traditional PDES event dependency.

Our contribution can be summarized as follow:

- an advanced memory management architecture, targeting Linux systems running on the x86 Intel-based architecture, has been designed and implemented in order to detect transparently the materialization of cross-state dependencies between concurrent simulation objects

- an advanced synchronisation mechanism is presented. It masks a traditional sequential simulation, where events are scheduled in a non-decreasing time-stamp and any object can access any valid memory area even if it is

logically owned by another simulation object, with a parallel one where events and cross-state dependencies are concurrently managed. It is totally hidden in the underling simulation platform with the purpose of making every aspect transparent to the application-level programmer. We call our scheme *ECS* (Event and Cross-State synchronisation).

This solution has been implemented and tested inside the ROme OpTimistic Simulator (ROOT-Sim) [2].

5.1 Intel x86-64 Paging [1]

For better understanding notions that we will introduce in the next sections, let us explain how the Intel x86-64 paging works.

By definition *paging* is the action of translating a linear address into a physical one with the purpose of using it for accessing memory or I/O devices. The computed physical address allows the system to verify if the current access is permitted (the address's access rights) and what is type of caching policy adopted for the pointed memory (the address's memory type).
Intel-64 processors supports three different types of paging:

- 32-bit paging
- PAE paging
- IA-32e paging

They differ with regard to:

- the size of the linear addresses that can be translated
- the size of the physical addresses produced by paging
- the granularity at which linear addressed are translated
- support for execute-disable access rights
- support for PCIDs
- support for protection key

[1] The notions of this section have been extracted from [47].

We are interested in IA-32e paging with page size of 4-KBytes.

5.1.1 Hierarchical Paging Structures

All mentioned modes use hierarchical paging structures. Each paging structure is composed by 4096 bytes containing 512 entries of 64 bits (8 bytes). Processor adopts a combination of those hierarchical structures with linear address that is being translated. Linear addresses can be divided in two parts: the first one states the entries of each paging level data-structures used to obtain the physical address of memory page that contains our data (page frame), while the second identifies the offset inside the previous identified memory page (page offset). Each one of 512 entries contains a physical address that points either the next paging structure level, namely it is *referencing* the other paging structure, or a page frame, and in this case we say that it is *mapping* a page.

Paging structures are named according to their use during the translation process.

Paging Structure	Entry Name	Paging Mode	Physical Address of Structure	Bits Selecting Entry	Page Mapping
PML4 table	PML4E	32-bit, PAE	N/A		
		IA-32e	CR3	47:39	N/A (PS must be 0)
Page-directory-pointer table	PDPTE	32-bit	N/A		
		PAE	CR3	31:30	N/A (PS must be 0)
		IA-32e	PML4E	38:30	1-GByte page if PS=1[1]
Page directory	PDE	32-bit	CR3	31:22	4-MByte page if PS=1[2]
		PAE, IA-32e	PDPTE	29:21	2-MByte page if PS=1
Page table	PTE	32-bit	PDE	21:12	4-KByte page
		PAE, IA-32e		20:12	4-KByte page

Figure 5.1: Paging Strucutures in the Different Paging Modes

Considering page size of 4-KByte, it translates 48-bit linear addresses to 52-bit physical ones mapping at most 256 TBytes of memory. In details:

1. PML4 table maps 256 TBytes, each of 512 PML4E points 512 GBytes
2. PDPT table maps 512 GBytes, each of 512 PDPTE points 1 GBytes
3. PD table maps 1 GBytes, each of 512 PDE points 2 MBytes
4. PT table maps 2 MBytes, each of 512 PTE points 4 KBytes

5.1.2 Translation

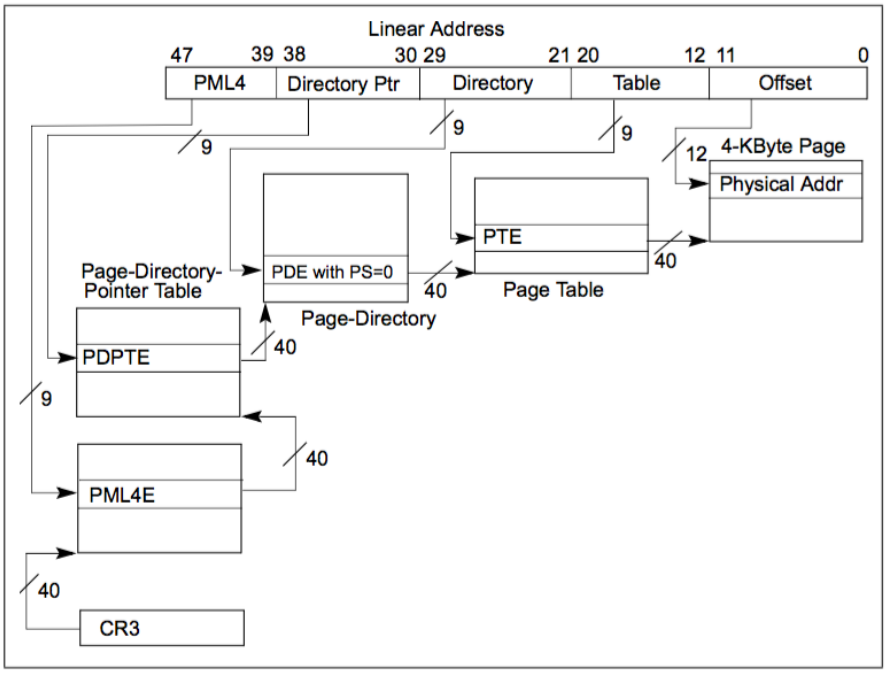

Figure 5.2: Linear-address translation to a 4-KByte page using IA-32e Paging

The translation takes place according to an iterative procedure. It starts by reading the value of CR3 register storing the physical address of the first paging-structure. The uppermost bits of linear address is used to select an entry within the first level data-structure. If the selected entry references another paging structure, the procedure continues with the next portion of linear address following the one just used. The translation terminates as soon as the selected entry map a page: in this case the lower bits of linear address represents the page offset within the identified page frame.

Only one case exists such that the translation terminates before a page frame

has been reached. This occurs when the procedure reaches a page-structure entry marked as "not present" or a reserved bit is set. In this case a *page-fault* exception is triggered meaning that either the translation is not allowed or simply it does not exist.

5.2 Event and Cross-State Synchronisation

This solution has been designed for targeting PDES platforms based on the multi-threading paradigm and exploiting NUMA facilities. In the last years, researchers have proofed how this can reach performances higher than simulators running as single-threads processes [43, 48, 49, 50].
Our case of study takes into account the situation where multiple threads dispatch whatever simulation object in a non-decreasing order based over the time-stamp of their simulation events. Technically, the execution starts by calling an ANSI-C function (event-handler) that takes as input informations like the *state base pointer*, that is the data-structure actually pointing all the dynamically allocated buffers for the current simulation object that can be accessed via pointers. This scheme traces the classical DES-style coding rules.

5.2.1 Cross-State Dependency Tracking

In this section we present the mechanism that we have designed and implemented for managing cross-state dependencies. This solution is totally transparent to the application-level programmer and it safeguards the benefits of multi-threading and NUMA paradigms.

Let us highlight again some technical aspects of our architecture. As described in Chapter 4, simulation objects see virtual memory in the form of *stocks*, namely a set of aligned pages that our allocator delivers on demand. The classical `malloc` service has been substituted with a custom allocator, therefore

each traditional malloc call is automatically redirected to our API. This returns memory buffers allocated via `mmap` POSIX API with size aligned to a power of two. However these buffers are not immediately allocated since they contain empty-zero pages, they will be allocated only upon the first read/write operation as the POSIX standard suggests.

With the idea of simplifying the cross-state detection, we have forced each simulation object to a priori allocate a single stock aligned to one entry of second level paging structure, namely a PDPTE. In other words, contiguous virtual-pages addresses composing a single stock are translated from virtual-to-physical by one PDPTE. It means that a single stock is exactly 1 GByte of memory, equivalent to 512^2 pages, and therefore handling multiple stocks within a single simulation object means managing multiple gigabytes of state at the same time. Figure 5.3 shows exactly this situation, starting from a PML4E we find a PDPT where each entry is reserved for a different LP: the virtual memory pointed by 0-th entry is hold by LP_x while the LP_y is set up over the 1-st entry.

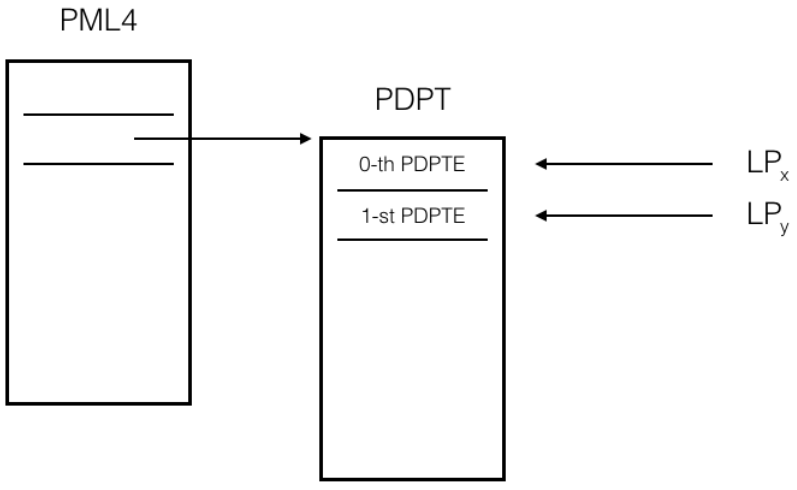

Figure 5.3: Example relation between LPs and Stocks

In order to track the materialisation of cross-state dependences, we need a mechanism that notifies us as soon as they appear. Namely, we want a solution for protecting the memory of each simulation object, in this way as soon as

an object accesses at least one area that is out of its control, say the state of another simulation object, the underlining memory management system informs us about the violation. Coping with this issue is not trivial, the complexity is in the fact that we are targeting Parallel-DES and therefore we concurrently dispatch simulation objects over different worker thread (WT). Our base case is:

1. LP_y schedules an event for LP_x containing within the payload also a pointer to memory area of its state. This pointer will be used by LP_x to directly handle information of LP_y.
2. LP_x executes the event and accesses the LP_y state via pointer

According to PDES, LP_x may be hold by WT_a while LP_y is handles by WT_b. The traditional memory protection of operating system, the segmentation-fault handling scheme, is not able to cope with our issue. Note that exactly the same page table is shared among all worker threads. Nonetheless, segmentation can detect accesses out of the state boundaries of a simulation object: thanks to `mprotect` POSIX API it is possible to change the state of page table, but this is not applicable because it hinders the concurrency. Other worker threads are actually dispatching different simulation objects and due to `mprotect` they will find the needed memory protected. The violation of this protection will generate memory faults which are not usually triggered. In other words, all objects, except for one, will trigger a memory fault as soon as they will try to access their own states instead of "remote" stocks.
Moreover, transparent code instrumentation is also not useful for dealing with our case study. It can be adapted but it will not perform. All read/write operations must be instrumented also if they turn out as not being cross-state dependencies.

The entire logic of our solution is implemented by a special device file driver added to the Linux Kernel via an external module. The simulator interacts with it using the following `ioctl` commands:

- `SET_ANCESTOR_PGD`
- `GET_PGD`
- `GET_FREE_PML4`
- `SCHEDULE_ON_PGD`
- `UNSCHEDULE_ON_PGD`

During the initialisation of LPs' memory, `GET_FREE_PML4` returns to the platform an address aligned with the first empty PML4E, the returned value is the initial address of the 1 GB of first LP that necessarily will be translated by one single PDPTE. All consecutive memory allocations for the other LPs will take place immediately after the returned value. For instance let us consider that `ioctl` returns i and we have n LPs, then the first n PDPTEs pointed by $i-th$ PML4E will contain all states of our LPs. If n is bigger than 512, then the platform calls one more time `GET_FREE_PML4` to retrieve the next free PML4E and the procedure continues as before.

The possibility of knowing exactly where are the stocks of each LP inside the page table is the key point of our memory management architecture. Any worker thread owns a completely new page table instantiated by `GET_PGD` at the beginning during start up of simulation platform, we name this new paging structure SIBLING_PML4 (henceforth we will make reference to the original paging data-structure as ANCESTOR). The `ioctl` command builds a copy of original PML4 setting to `NULL` those PML4Es that translated our stocks. The `NULL` value represents our protection mechanism: if during the paging of whatever logical address we bump into a `NULL` entry it means that the related translation does not exit and a *page-fault* exception is triggered, exactly as we said in Section 5.1. Therefore we do not grant the possibility to reach the lower level paging structures, blocking any possible access to the previously allocated stocks.

When a worker thread WT_i has to execute an event for simulation object x its SIBLING_PML4 is updated as follow (i.e. we say that we have "opened" the

memory of x for the worker thread WT_i):

1. a new PDPT, namely a SIBLING_PDPT, is instantiated
2. the entries related to the stocks of x inside the SIBLING_PDPT are set up copying the value from the corresponding ANCESTOR_PDPT
3. the associated PML4E inside the SIBLING_PML4 is populated with the physical address of new PDPT instanced at STEP 1

In this way WT_i can compute the correct translation from virtual to physical of those addresses that point the stocks of x. Obviously, in case some pages of x are not present it means that are swapped-out pages. Note that we have opened to WT_i only the memory of x while all others stocks owned by other simulation objects are not accessible yet.

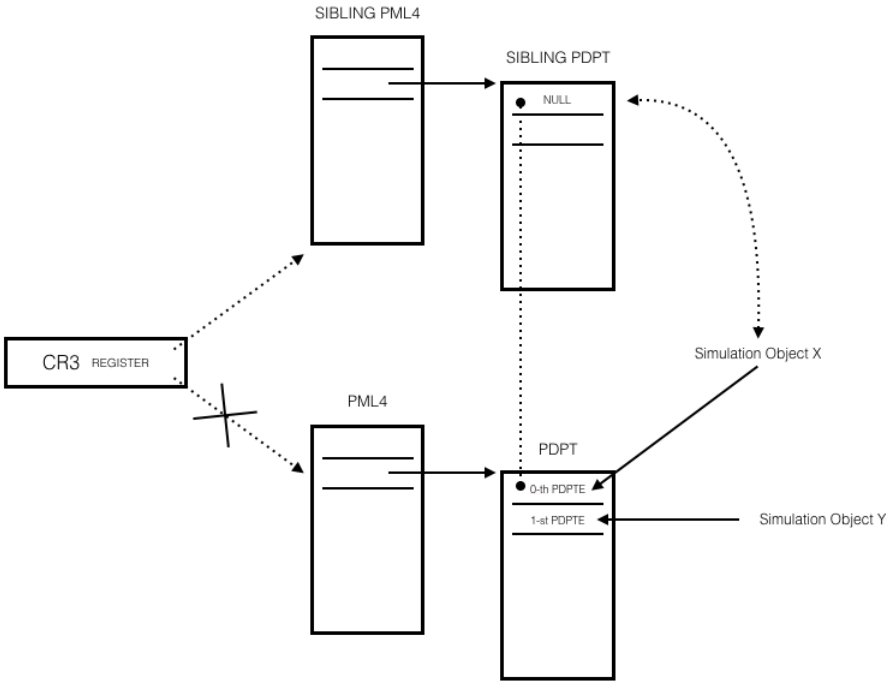

Figure 5.4: Memory Stock of LP_x is opened to the current Worker Thread

Figure 5.4 shows exactly the situation explained above: the executed steps are marked with dotted lines. Again the x's stock is handled by the $0-th$

entry. This operation is computed thanks to the command `SCHEDULE_ON_PGD`: after the execution of previous 3 steps the value of page table pointer register, namely `CR3` in X86_64 processor, is substituted with the physical address of SIBLING_PML4. In this way the current WT_i enters into *simulation-object mode*. Let us stress again that only the entry associated to the dispatched object has a value different from `NULL` and therefore the stocks of x are the only ones that can be accessed, regarding other stocks there is no way to reach them because all associated entries are set to `NULL`. Note that changing the value of `CR3` is an onerous function because the logic of x86_64 processors firmware automatically flushes the `TLB`.
The possibility of concurrently dispatching and executing different simulation objects while tracking whenever any object tries to access memory out of its boundaries is given by the presence of different PML4 tables for each thread: we have a different SIBLING_PML4 per thread. Our architecture is based on the idea that the ANCESTOR paging data-structure are already filled up with all needed information. A simple `mmap` does not leave data-structures in this situation, it just set memory to empty-zero value. In order to overcome this default behaviour, after mmapping the required memory our architecture writes also an entire page of `NULL` values forcing Linux Kernel to initialise the whole chain of structures used to manage our stocks. This guarantees the existence of each PDPTE associated with our stocks and used to update the SIBLING structures during the dispatch of simulation objects.

`UNSCHEDULE_ON_PGD` is the dual command of `SCHEDULE_ON_PGD`. As soon as a simulation object completes an event execution, the related worker thread calls the former `ioctl` command. This simply rewrites the original pointer to the ANCESTOR_PML4 inside `CR3` and then it destroys the SIBLING_PDPT used to access the stock, in this way during a new despatch a new `SCHEDULE_ON_PGD` is re-triggered and the architecture has just to rexecute the steps presented above: every situation is reduced to a single case easier to manage. This execution

brings the simulation object back to the *platform mode*.

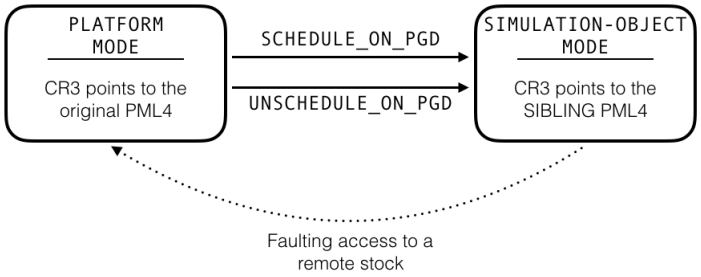

Figure 5.5: State diagram of simulation object according to the value of CR3

We have not discussed how the cross-state dependencies are tracked yet. As we said before, as soon as a worker thread tries to access any stock that is not handled by the current dispatched object a *page-fault* exception is triggered since the related entries of PDPT contain `NULL` values. The classical segmentation-fault management leads to reallocating the entire chain of paging data-structures related to the interested stock since the stocks have been already allocated via `mmap`. It means that for the same virtual page there might be multiple page tables entries that point it. We want to avoid this behaviour since it can not be directly handled by Linux Kernel without a custom patch. To cope with this issue, we have created a custom version of page-fault handler that is substituted to the original one during the initialisation of our file driver changing the corresponding value inside the IDT table that can be accessed via `IDT` register. Our version of page-fault handler knows what are the entries of PML4 table interested by our stocks, therefore upon fault it checks whether the problem is related to our PML4Es; if so, it verifies whether the entry is already present inside SIBLING PML4 and in the negative case this is a cross-state hence the process ends; whereas when the PDPTE is present it controls whether the fault is confined to the PDPT so that it is a cross-state dependency, otherwise it is a normal memory fault and the normal `do_page_fault` of Linux Kernel is called. In case of cross-state dependencies, our handler identifies the involved stocks and the resulting PDPTEs. This information is stored inside the stack by the handler and the control is given back to the platform. After the control

comes back to the platform, architecture retrieves the new information from the stack, updates the needed data-structures and calls `UNSCHEDULE_ON_PGD` (see Figure 5.5). In this way we can synchronise the simulation objects involved in the dependencies via our synchronization scheme that will be depicted in the next section. During this management we pass from *user-mode* to *kernel-mode* and then again to *user-mode*. In the past such behaviour was exacting for high performance computing, but nowadays thanks to the low latency provided by machine instructions `sysenter` and `sysexit` the prohibitive costs have decreased by one-forth making our implementation possible and faster. These machine instructions target operating systems without segmentation and with a memory model that is flat.

We base our architecture on the possibility of changing the value of `CR3` and leaving it unchanged until the conclusion of dispatched event. During the re-dispatching of a thread, the scheduler retrieves the pointer of current thread page table that is actually going into CPU from the memory context. This memory management information stores the address of original page table, namely the ANCESTOR one, but since we do not have any assurance that a dispatched event has been completed with just one run we need a mechanism to refill `CR3` with the SIBLING PML4 if necessary; in this way the worker thread is allowed to access only the needed stocks while the others can be still protected or concurrently dispatched along other threads. In order to deal with this issue, we have designed another kernel module that patches the Linux scheduler, adding a new function at the end of schedule that simply checks if a special function-pointer is not `NULL`, if so it calls our file driver that sets up `CR3` with proper value. This function exploits some meta-data initialised with `SCHEDULE_ON_PGD` command to understand if `CR3` must be updated with the related SIBLING PML4. Obviously, if the pointer is equal to `NULL` the reschedule action works according to the standard behaviour. In this way our architecture can coexist with the traditional schedule of Linux Kernel.

5.2.2 The Event Cross-State Synchronization Scheme

A new synchronisation scheme has been designed in order to synchronise the simulation objects involved in the cross-state dependency, it exploits the memory management described in the previous section. While it allows each simulation to process its events according to a non decreasing time-stamp order for entailing correctness inside a PDES platform, it guarantees also that a cross-state dependence occurred at time t will find the other involved stocks at the same simulation time t, as if it had been materialised in a sequential environment where simulation objects advance always in a non-decreasing time-stamp order. To achieve our goal, we have augmented the number of possible states in which a simulation objects can pass though giving the possibility of returning back the control to the platform even if the event execution is not yet completed (interrupt-driven scheme), and we have introduced a special class of events called *rendezvous* that can be triggered only by the platform in order to temporarily disable involved simulation objects and align them according to simulation time at which the dependencies has been materialised, since they are platform-generated events they do not have any associated processing rule at the application level, let us stress again that our solution is totally transparent to application-level programmer. These two innovations allow us to improve the PDES execution model with some concept of Transactional Memory models: we have made read and write operations serialisable across multiple stocks according to the logical time at which their occur.

As the name of scheme suggests, a cross state dependency terminates together with the event that materialised it. Therefore all new data-structures added to the platform in order to handle cross-state dependences will be emptied after the conclusion of each cross-state event.

Each simulation object has been associated with a cross-state dependency set that stores all identifiers of simulation objects towards which the current

object has materialised cross-state during the execution of an event. We refer to set of x as CSD_x. Upon dispatching of new event, CSD is initialised as empty and it will be updated as soon as the memory management observes a cross-state dependence. Let us consider that simulation object x is trying to access either in read or write mode the stock of y during the execution of its event e_x since it has received a memory pointer from y inside the payload of its event e_x. A cross-state dependency is materialising. Our page-fault hander is called, and the needed parameters are given to the platform. Handling of ECS takes place in accordance with the following algorithmic steps:

1. object x passes through a blocking state and therefore the execution of its event e_x is temporarily blocked
2. a unique identifier of rendezvous $rvid(e_x)$ is computed and added to the payload of e_x
3. a rendezvous event e_y^{rv} is sent to object y, it has the same rendezvous mark and the same time-stamp of e_x

$$ts(e_x) = ts(e_y^{rv}) \qquad rvid(e_x) = rvid(e_y^{rv})$$

Rendezvous events are treated as classical events, therefore they are inserted into the event list of the receiver object, in our case y. It means that a rendezvous event may be a straggler event in the case that the simulation time of the receiver is bigger than the time-stamp stored inside the payload of event. This case may be highly possible since we are targeting speculative environments. Another important aspect is that rendezvous events are handled as application-level event without a rule defined by the programmer, their behaviour is hidden inside the platform and generates results at platform level as a classic event without updating the receiver state.

When e_y^{rv} becomes the event of y with lowest time-stamp, the following algorithmic steps are actuated by ECS in order to handle it:

1. object y passes through a blocking state
2. a rendezvous acknowledgement event e_x^{rva} is sent to object x with the same

rendezvous mark of e_x but without time-stamp

$$rvid(e_x) = rvid(e_x^{rva})$$

On the other side, as soon as x receives e_x^{rva} ECS actuates the following algorithmic steps:

1. the identifier of y is added to CSD_x
2. x comes back to the ready state and it will be eventually dispatched over some worker thread and it will try to complete the previous interrupted event e_x

Object y will be blocked until x finishes e_x, since we want to allow x to read and/or write stocks of y in order to complete the operation that originates the cross-state dependency. Upon re-dispatching object x the underling memory management uses CSD_x for opening to x not only its stocks but also the one owned by y; in order to do so we pass to the `SCHEDULE_ON_PGD` command the set $CSD_x \cup x$, that in our example states x and y.
All showed steps can be iterated in case object x materialises other cross-state dependencies towards multiple LPs. According to our memory management architecture and our synchronization protocol, any access to stocks owned by simulation objects whose identifiers are stored inside CDS will not lead to new ECS memory fault.

After the ending of e_x, ECS informs all objects whose identifiers are stored inside CSD that the cross-state dependency is finished and therefore they are no longer blocked and after resuming they can go ahead with their event. To achieve this result ECS executes the following algorithmic steps:

1. for each identifier k stored inside CSD_x, a rendezvous unblock event e_k^{ub}) is generated with the same rendezvous mark oof e_x but without time-stamp

$$rvid(e_x) = rvid(e_x^{rva})$$

2. upon the delivery e_k^{ub}, the receiver gets back to ready state and it will be eventually re-scheduled

This section and the previous one describe in details our solution but they omit some details in regards to correctness and progress. We also need to explain in detail how to handle rollbacks and anti-messages since we are targeting speculative environments.

Correctness

ECS introduces new relations between at least two simulation objects which previously where not possible: it allows object x to read and/or write the stocks of other objects as in-memory transactions, therefore x must find all needed stocks, its own and the ones handled by the other involved objects, at the very same simulation time. In other words, if during the processing of an event e_x a rendezvous event is inserted into the system thus it means that from now on the involved objects are causally related and therefore as soon as one of them rollbacks at time preceding the cross-state materialisation our protocol has to take into account this relationship. Particularly, a rollback of one simulation object induces the rollbacks of all simulation objects involved into the cross-state. The possible situations are two, if object x rollbacks at time $t' < ts(e_x)$ and e_x had generated a e_y^{rv}, we need to rollback also y because x during its execution may have performed some updates on the stocks of y; if y has to rollback at time $t' < ts(e_y^{rv})$ we need to rollback also x since x may have used some information handled within the state of y to update its local state affecting the outcome related to the execution of e_x.

To cope with this issue we implement the following scheme. To solve the problem generated by the rollback of e_x, we simply send an anti-event related to the rendezvous event e_y^{rv} that will drive the classic annihilation phase that leads to rolling back y to the latest processed event with time-stamp smaller than $ts(e_x)$. This is possible since the rendezvous event was added into the event list of receiver. If instead y, that actually is the simulation object that received the

rendezvous event, has to rollback at a simulation time smaller than $ts(e_y^{rv})$ we need to annihilate any possible change made by x using the simulation state of y. This is done by means of a special anti-message that will be sent to x. When e_x will be dispatched again after rollback, a new marked rendezvous will be generated therefore no discrepancy will occur and no cycles will be generated by annihilation process.
With respect to the other ECS events, such as acknowledgements and unblocks, they are not added into the event lists of the receiver and therefore they can be simply discarded without requiring a rollback scheme. Considering FIFO channels between simulation objects, these events can be rejected if the rendezvous mark stored inside the event payload is different from the one owned by simulation object that receives the event.

Progress

The classical issues that can hamper performance inside a speculative PDES environment may be generated also from our ECS scheme, therefore we need to take care to avoid deadlocks, livelocks and domino-effects during the rollback phase.

Whenever an object x generates a rendezvous event at time t_1 directed to z while it is waiting for y to reah time t_3 for a rendezvous between y and z, while y in turn is expecting that x reaches time t_2 for a rendezvous between x and y, this leads the system into a deadlock condition. A rendezvous cycle is generated: in order to continue its ECS execution, object with minimum time-stamp waits the rollbacks of other simulation objects while in turn they are in a blocked state waiting for the unblock of one of them and the object with minimum time-stamp.

In order to avoid this scenario, we augment our simulation scheme with the following rule: whether a simulation object x has to rollback while it is blocked

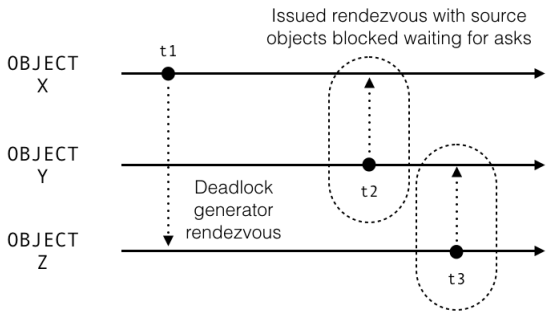

Figure 5.6: Example where a rendezvous event generates deadlock

due to a rendezvous event generated by processing event e_x, before setting it back to a ready state, event e_x will be squashed and relative anti-event will be sent into the system to inform all involved objects that event e_x has been forced to finish and a rollback is needed (See Figure 5.6). From the point of view of involved objects, a rollback is needed because x may have modified their state, therefore we must guarantee that after resuming all simulation objects have a coherent state. According to our synchronization scheme, using anti-message to annihilate a rendezvous event is safe since the involved objects are all blocked, therefore they can handle it immediately.

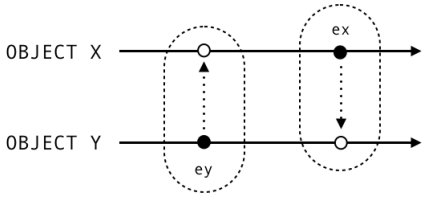

Figure 5.7: Example where rendezvous events generate livelocks

Live-lock is another classic issue that we have to take into account. Simulation objects that materialise circular cross-state dependences will surely enter in a livelock condition. Figure 5.7 shows exactly this situation, object x processes e_x at simulation time $ts(e_x)$ and it raises a cross-state dependence towards y, meanwhile y is executing e_y at simulation time $ts(e_y) = ts(e_x)$ that in turn generates a rendezvoud event toward x. This situation may cause a cycle of

rollbacks that may be repeated indefinitely. This issue can be solved by a simple priority management scheme for simultaneous events extended to rendezvous events. We need a mechanism to identify concurrent events that are causally related. If the presence of two simultaneous events e_x and e_y with e_y causally related to e_x, namely $ts(e_y) = ts(e_x)$ such that $e_x \rightarrow e_y$, we must extend this property also to any rendezvous event generated by x: any rendezvous event e_y^{rv} generated by x towards y during the e_x processing will be in turn causally related to e_y, namely $e_y^{rv} \rightarrow e_y$. In this way any possible conflict materialised by simultaneous events will be serialised according to their causality relations. Issues related to simultaneous events are a general problem of speculative PDES environment. In the last years, they were widely studied and some solution have been provided in order to tie-breaking simultaneous events that can be integrated with the just showed scheme.

The last issue that we have to take into account is the domino effect. We recall that a cross-state dependency between x and y means that x will directly manage the state of y as soon as the latter will grant it the access. In order to show to x a consistent snapshot of state of y, y has to be at simulation time exactly equal to x. This condition is assured thanks to our synchronisation scheme. Due to a rollback of x at simulation time smaller than the simulation time of event e_x that raised cross-state dependency, x may execute e_x in silent execution. Note that during the coasting-forward phase no one of previous already sent messages will be resent since executing in silent execution means reaching the needed time starting from the first available state with time smaller than the straggler event that caused rollback. In other words, simulation objects will reprocess some events that have not been annihilated according to the classic rollback handling for speculative PDES environment in the case of sparse state saving. Hence x may access the state of y without synchronisation, in this situation two different issue may be verified. If both x and y are concurrently dispatched over two different worker threads, the former could concurrently han-

dle the same memory area of the latter; meanwhile y is executing on the other worker thread. Instead, if y is not dispatched while x is executing, the latter may access the state of former while the former is at simulation time different from the one of x thus a violation of coherence will be materialised losing the correctness of simulation: x will observe a state of y that may be either older or earlier of its current simulation time. If x rollbacks the only way to ensure consistency is to rollback also y because they are causally related due to their cross-state dependency. It is easy to show that the rollback of y may trigger also the rollback of another object z due to a cross-state dependency as well and a so called domino effect may be generated. With the purpose of avoiding this incorrect behaviour our scheme forces a new log, exactly at the end of each cross-state event, to the logs taken by the sparse state saving policy thus it will be impossible that during coasting-forward our simulation objects will dispatch cross-state events: no rendezvous generating event will ever be in the sequence of events between two subsequent logs of the same simulation object.

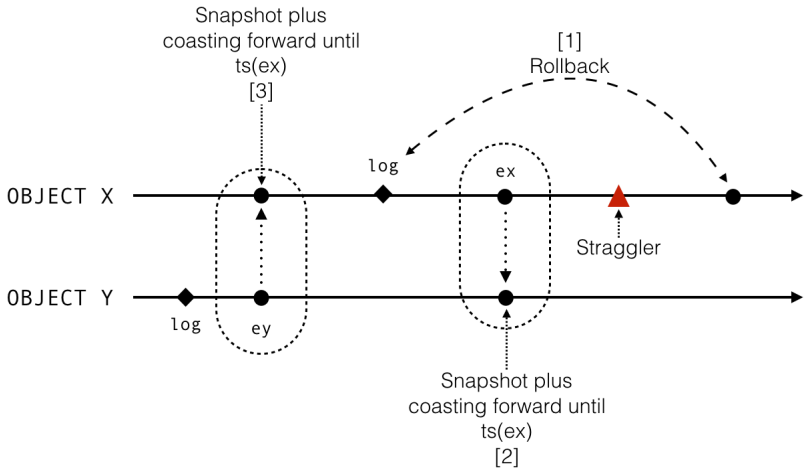

Figure 5.8: Example where rendezvous events generate cascading rollbacks

Another solution for our issue is taking logs at the end of each correctly processed event, but it is not practicable because it hampers performance of our system.

5.2.3 Integrating State Sharing Policies with NUMA Oriented Support

The ROme OpTimistic Simulator (ROOT-Sim) is our test-bed environment. The environment described in Chapter 4 has been augmented with some new functionality in order to implement our advanced memory management architecture as well as our advanced synchronisation scheme.

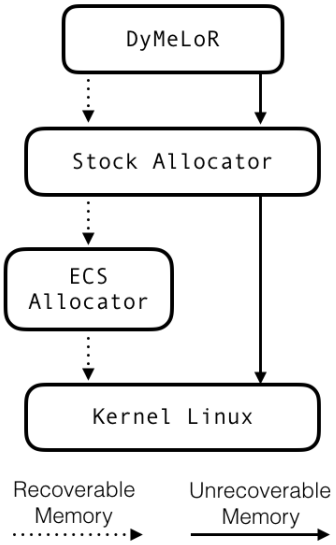

Figure 5.9: New layer for allocating 1GB of contiguos memory

In order to handle the new memory constraints we have added a new layer below the Stock Allocator (see Chapter 4) of ROOT-Sim called `ECS_allocator`. Inside our platform, the memory demands are taken into account by the DyMeLor open source allocator [29, 51] cooperated with the Stock Allocator (see Chapter 4), they manage two types of memory: unrecoverable and recoverable. The former is directly asked to the Kernel, while the latter is demanded from our new allocator. We recall that our solution for properly working needs that each LP owns exactly 1 GBytes of aligned memory pages. `ECS_allocator` has been created for exactly fitting this purpose. During its initialisation, DyMeLor initialises the Stock Allocator that in its turn triggers the `ECS_allocator` by

calling `allocator_ecs_init`. This function forces the reservation of 1GBytes for each simulation object by calling the `mmap` API of Linux Kernel, writing a `NULL` byte into one single stock's page, namely the first one, and concludes the procedure by populating our new data structure `lp_mem_region`. A single call to `mmap` API is not enough to allocate the needed 1GBytes because the amount of memory that can be obtained with a single call is limited to at most 512 MBytes thus we have to call it twice.

Algorithm 2: allocator_ecs_init

```
1:  procedure ALLOCATOR_ECS_INIT(int n_prc)
2:      size ← (512 * 512 * 4096)/2                          ▷ 512MB
3:      for i ← 0, n_prc do
4:          index ← IOCTL(GET_FREE_PML4)
5:          addr ← GET_MEM_ADDR(index)
6:          for y ← 0, 511 do
7:              lp_mem_region[i].base_pointer ← MMAP(addr,size)
8:              addr ← addr + size
9:              MMAP(addr,size)
10:             addr ← addr + size
11:             i ← i + 1
12:         end for
13:     end for
14: end procedure
```

Before calling mmap, the platform calls `GET_FREE_PML4` for retrieving the first free PML4E, starting from its index we compute the initial address aligned with the first PDPTE pointed by the retrived PML4E. `GET_FREE_PML4` simply scans ANCESTOR PML4 locking for the free entry, as soon as it finds the index, it updates a local vector called `dirty_pml4` that tracks what entries of PML4 level are under control of our allocator. Populating the initial page of each stock is crucial: as soon as the Linux Kernel accesses an empty-zero memory area it allocates the whole chain of paging data-structure that we will exploit for managing cross-state events.

The new data-structure that tracks of recoverable memory of each simulation object is composed by the following two fields:

- `base_pointer`: identifying the initial address of corresponding stock, namely the base of allocated memory associated with the LP state. `NULL` has been used as the default initialisation value.
- `brk`: setting the end of data segment already used by the current simulation object.

As of `malloc` library, we implement a classical per LP pre-allocation strategy. This is done in order to have memory ready for serving the on demand requests of DyMeLor API which tries to improve memory locality for the LP state with the purpose of optimising checkpoint/restore operations. Hence at the end of `allocator_ecs_init` we have allocated an array of as many entries as the number of simulation objects where each entry contains exactly one `lp_mem_region` structure, and we have allocated as many Giga bytes as the number of simulation objects.

As soon as DyMeLor needs a new `malloc_area` for recoverable memory we identify which LP is calling the API and then we redirect the request to `ECS_allocator` that will try to deliver the new memory retrieving it from the pre-reserved Giga. This operation is computed by `get_memory_ecs` function. Starting from the caller's `lid`, the allocator identifies the associated `lp_mem_region` and returns memory only if the value of `brk` plus the required size does not exceed the preserved 1 GBytes. In the positive case, it returns the initial pointer of new memory and increments the `brk` fields of size passed as input to the request, while in the negative case it notifies to the upper level that the required memory is not available.

The last API that we have introduced is `get_base_pointer`. In order to trigger the `SCHEDULE_ON_PGD` command, we have to populate the data-structure that it takes as input, namely `ioctl_info`. It is composed as follow:

```
1 struct ioctl_info{
2         unsigned mapped_processes;
3         ulong callback;
4         int ds;
5         void** objects_mmap_pointers;
6         int objects_mmap_count;
7 }
```

`objects_mmap_pointers` contains the addresses of stocks that we want to "open" to the simulation object which we are dispatching. Those addresses are retrieved by calling `get_base_pointer`: giving the `id` of the simulation object that holds the stock which we are interested in `get_base_pointer` returns the value of `base_pointer` field stored inside the `lp_mem_region` data-structure associated with the engaged LP.

Together with the innovations introduced in the management of recoverable memory, we modify the procedure for dispatching simulation objects. In order to launch the execution of simulation events, first of all platform has to trigger `SCHEDULE_ON_PGD` command: it is in charge of checking if the ANCESTOR and the corresponding SIBILING pgds differ within those entries that are not taken into account by `ECS_allocator`, then our file driver finds what entries of SIBLING paging data-structure must be populated for "opening" memory to the dispatched LP according to the addresses provided inside `objects_mmap_pointers` field. We recall that, ANCESTOR and SIBLING paging structures differ within just entries that point to the states of our LPs, namely within entries that we use to implement our protection mechanism, while the former has these entry correctly populated, the latter contains just `NULL` value. In the end, `SCHEDULE_ON_PGD` notifies to the platform that SIBLING pgd's pointer must be loaded inside CR3 register for moving the execution over the parallel view.

`UNSCHEDULE_ON_PGD` is the dual of `SCHEDULE_ON_PGD`. This command is triggered by the platform after the conclusion of each simulation event. It loads inside CR3 the ANCESTOR pgd's pointer and then empties by writing `NULL` val-

Algorithm 3: SCHEDULE_ON_PGD

```
    procedure SCHEDULE_ON_PGD(LpMemRegion MRS, int wt)
2:      CHECK_DIFF(SIBLING[wt], ANCESTOR[wt])
        for mem_reg ∈ MRS do
4:          entry = GET_PML4E(mem_reg.address)
            OPEN(entry, SIBLING[wt])
6:      end for
        LOAD_CR3(SIBLING[wt])
8:  end procedure
```

ues all those PDPTs previously instantiated by `SCHEDULE_ON_PGD`. Substituting CR3 value at run-time is a safe operation inasmuch the two paging structures differ only in terms of those entries that map stocks, in other words the two structures are equal except for entries saved inside `dirty_pml4` vector enhanced in the ANCESTOR but not in the SIBLING one. The freeing of SIBLING PDPTs are needed to simplify the code of `SCHEDULE_ON_PGD`: there is no difference if a worker thread calls this command for the first time or for the n-th time, it always must instantiate all entries related to the given stocks without taking care if some of them were enhanced beforehand.

As stated before, "opening" memory only at the start time and closing it at the end of execution is not enough: a worker thread may be de-scheduled even if its cross-state event has been not yet completed. For this reason, we need a function, that during the re-dispatching of a worker thread that has not been concluded yet, reloads its current simulation event with the proper value inside CR3 register, namely the pointer to the SIBLING pgd. Another kernel module is created for coping with this issue: `schedule_hook` that stores inside one of its parameters the function pointer of `load_sibling` method. At run-time after the last step of `schedule` function of Linux Kernel, `schedule_hook` adds a call to `load_sibling` that checks if the current Linux thread is one of the worker threads of our platform and if it is necessary to refill CR3 register with its SIBLING PGD, in this way the worker thread will be allowed to access only the stocks opened after calling `SCHEDULE_ON_PGD` while the other

stocks remain protected or concurrently dispatched along other threads. On the other hand, there is no problem if the system re-dispatches a worker thread that has already concluded its event, because during the de-scheduling Kernel automatically saves inside the memory context the value of ANCESTOR pgd and therefore it automatically restores it inside CR3.

We have not explained how our solutions tracks event cross-state materialisation yet. Another Kernel module is created for overriding the classical page-fault handler of Linux Kernel: instead of calling `do_page_fault` our handler simply calls `root_sim_page_fault` after the materialisation of any memory fault. `root_sim_page_fault` checks whether the fault is related to one of our `dirty_pml4`, if this is not the case it passes the control to the traditional page-fault handler otherwise a cross-state dependency is just raised and therefore starting from the value of CR2 register in which the firmware stores the address that has caused the memory fault, it identifies which is the simulation object hit by a cross-state dependency. These actions are performed at kernel-level where all informations needed to finalise the synchronisation between the involved LPs are not available: at ring-0 we can identify which entry of PML4 is involved while the data-structures that must be updated for keeping track of cross-state relation are stored at user-level. Therefore we need to give back the control to our platform but not before storing into the stack the information retrieved at kernel-level. As usual inside Linux Kernel, we use the `stack` for passing parameters from kernel to user level and vice versa since this is the only common area between these two levels. At the end of `root_sim_page_fault` execution, the `stack` contains (see Figure 5.10):

- the `pid` of current worker thread
- the `id` of target LP
- the instruction value register used for reactivating the execution after the finalisation of our synchronisation protocol

Our custom page-fault handler concludes its execution by substituting the IR register with the function pointer of our `ECS_stub`, in this way the control

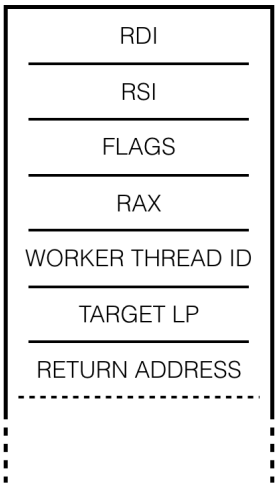

Figure 5.10: How ROOT-Sim page-fault handler left the stack

comes automatically back to the user-level. The `ECS_stub` is an X86 Assembly stub with just one goal: it has to prepare our CPU for calling `ECS_handler`: it arranges registers by putting inside the values written within the stack by the Kernel and it finally calls the `ECS_handler`. This last handler is the starting point of our synchronisation protocol: it blocks and aligns the involved objects in terms of simulation time. How the involved objects are really blocked is described in the next paragraphs. The entire work flow is described by Figure 5.11.

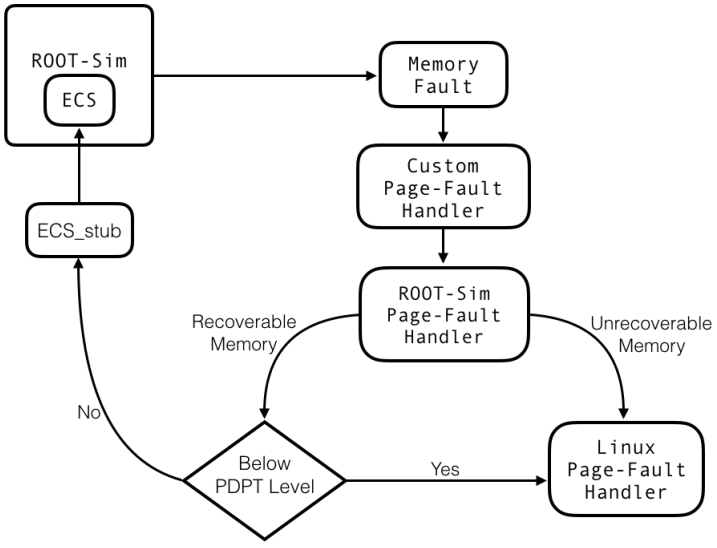

Figure 5.11: Excution flow of ESC traking

Our new synchronisation protocol requires some innovations as well, the two main changes are

- the introduction of three new LP states for tracking event cross-state dependence between involved simulation objects
  - `READY_FOR_SYNCH`
  - `WAIT_FOR_SYNCH`
  - `WAIT_FOR_UNBLOCK`
- the creation of four new types of messages used by simulation objects for synchronising their operations
  - `RENDEZVOUS_START`
  - `RENDEZVOUS_ACK`
  - `RENDEZVOUS_UNBLOCK`
  - `RENDEZVOUS_ROLLBACK`

The state machine of our simulation object is evolved as shown in Figure 5.12. To the states presented in Chapter 4, we have added the notion of blocked state: states that cannot be considered by the scheduler. They are:

- `WAIT_FOR_SYNCH`
- `WAIT_FOR_UNBLOCK`

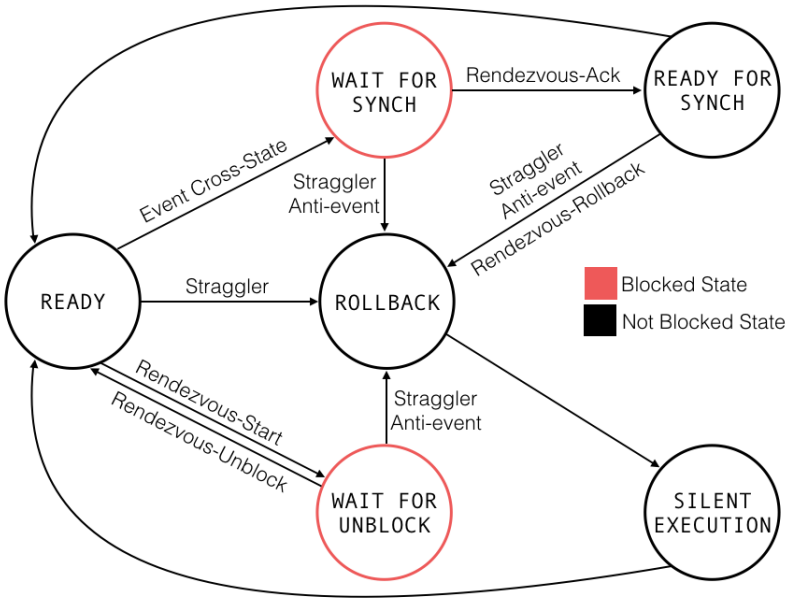

Figure 5.12: State diagram of simulation object

Henceforth, the ROOT-Sim scheduler will despatch LPs by taking into account both their local virtual time for satisfying the "select time-stamp first" condition and their state as well: it will select the LP with the minimum time-stamp that holds a non blocked state. In this way each simulation object that is involved in a cross-state dependency will be dispatched only at the end of our synchronisation protocol according to this rule: the LP that has been hit will become schedulable only after that the one that raised the dependency has already terminated.

Algorithm 4: SCHEDULER

```
    procedure SCHEDULE
        LP ← LPS[0]
3:      for i ← 1, n_prc do
            if LPS[i].lvt < LP.lvt ∧ ¬isBlocked(LPS[i]) then
                LP ← LPS[i]
6:          end if
        end for
        return LP
9:  end procedure
```

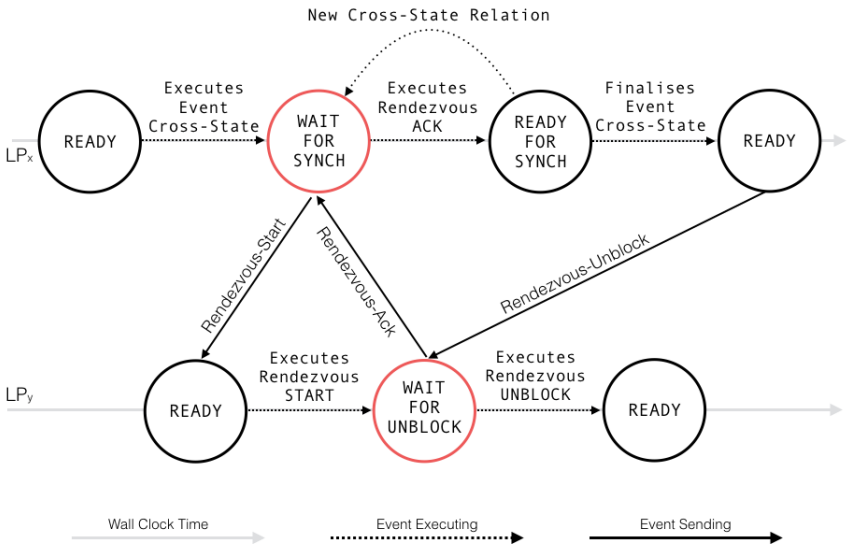

Figure 5.13: Evolution of LP states during ECS handling

In details, when an LP hits another simulation object, the hitter is set as `WAIT_FOR_SYNCH` and sends a `RENDEZVOUS_START` to the stricken LP at the same

simulation time of event that caused the cross-state, a procedure actuated by the `ECS_handler`. As soon as the LP that has been hit by cross-state dependence reaches the correct simulation time for processing the `RENDEZVOUS_START`, it passes into the `WAIT_FOR_UNBLOCK` state and sends back to the striker the `RENDEZVOUS_ACK`. Inside the payload of these control messages, there is a special field called `rendezvous_mark`, generated inside the `ECS_handler`, that is used to distinguish the control messages of a rendezvous execution from messages of another: we recall that in case of rollback a rendezvous execution may be either discarded or re-executed, in the latter case the new execution will take place with a different `rendezvous_mark` from the one that was used previously. When the LP that has partially executed the cross-state event receives the `RENDEZVOUS_ACK`, it switches to the `READY_FOR_SYNCH` state: meaning that as soon as it will become the LP that handles the event with the minimum time-stamp inside its thread, the scheduler will dispatch it for trying to finalising the cross-state event. If this LP is able to conclude the event, in other words if no new cross-state dependencies are materialised, it comes back to the `READY` state and sends to the hit objects the `RENDEZVOUS_UNBLOCK` messages. On the other hand, when hit objects process the `RENDEZVOUS_UNBLOCK`, they come back to the `READY` state. Let stress again that all rendezvous messages associated with the same simulation event have the same rendezvous mark and the same time-stamp of event that has caused the relation.
Since we have introduced the notion of blocked state, we had to modify also the point at which messages are executed: a blocked LP will never transit through the schedule function until it is bocked, hence it is impossible for it to execute control messages inside the schedule function rather than normal simulation events. If an LP is blocked, it will manage the control messages meanwhile it processes the bottom-half queue. In details:

Taking care of control messages during the processing of bottom-half does not violate the "smallest time-stamp first" condition because thanks to the ren-

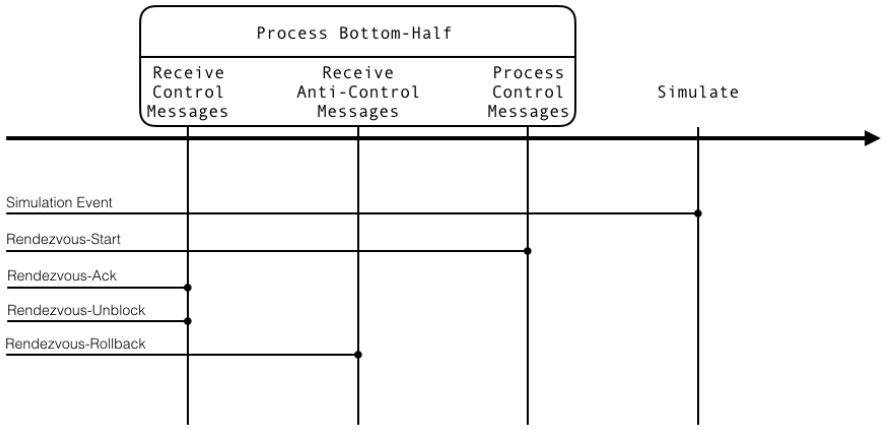

Figure 5.14: Processing flow of controll messages

dezvous mark we are able to recognise the rendezvous messages that we are looking for and we are also sure that they are exactly next logical events for the blocked LPs: no message inversion is created. In case of LP that has already processed the `RENDEZVOUS_START` the only possible next message in terms of logical time is the corresponding `RENDEZVOUS_UNBLOCK`, while for the one that has sent the `RENDEZVOUS_START` the only possible next message is the `RENDEZVOUS_ACK`.

Obviously the synchronisation procedure is not free from rollbacks, we are targeting speculative environments and for this reason the protocol must take into account rollbacks. The message pattern presented above has one advantage: from the point of view of a stricken object processing `RENDEZVOUS_START` and sending the related `RENDEZVOUS_ACK` is an atomic action as well as processing `RENDEZVOUS_ACK` and the consequently `RENDEZVOUS_UNBLOCK` by the striker is an atomic action (i.e. considering that no other cross-state relations are materialised for the same simulation event): it means that it is not possible to take logs between either `RENDEZVOUS_START`-`RENDEZVOUS_ACK` or `RENDEZVOUS_ACK`-`RENDEZVOUS_UNBLOCK`, in other words it is impossible rolling back inside one of these couples, therefore they can only roll back

1. before sending `RENDEZVOUS_START` that is the same simulation time pre-

vious to the processing the cross-state event

2. after sending `RENDEZVOUS_START` that is the same simulation time previous to the receiving `RENDEZVOUS_ACK`
3. after sending `RENDEZVOUS_UNBLOCK` that is the same simulation time of the end of cross-state event
4. before receiving `RENDEZVOUS_START`
5. after sending `RENDEZVOUS_ACK` that is the same simulation time previous to the receiving `RENDEZVOUS_UNBLOCK`
6. after processing `RENDEZVOUS_UNBLOCK` that is the same simulation time of the end of cross-state event

According to the definition of silent execution actuated during the coasting forward phase, rollbacking situations described in 2 and 5 must be avoided. During the re-processing of events in silent execution the involved simulation object does not re-send messages that were already sent during the previous normal processing otherwise it duplicates them and furthermore their receivers have already replayed to them. It means that since inside its messages queue the rollbacking object already owns the messages required to complete the event cross-state synchronisation, it will be sure that the involved objects are synchronised with it but instead they are in another simulation time: this behaviour destroys the coherence of our simulation. Hence we do not take logs in these two situations as well as executing cross-state events in silent execution. The latter situation is guaranteed by the fact that at the end of each cross-state event any involved simulation object takes a log. Due to a straggler event they can come back either before the simulation time of cross-state event or immediately after it. Adding these new logs to those that are already taken by the sparse state saving policy of our platform avoids also the possibility of generating domino-effect due to event cross-state rollbacking. The following scheme explains how logs are taken:

During an event cross-state handling the possible situation in which our

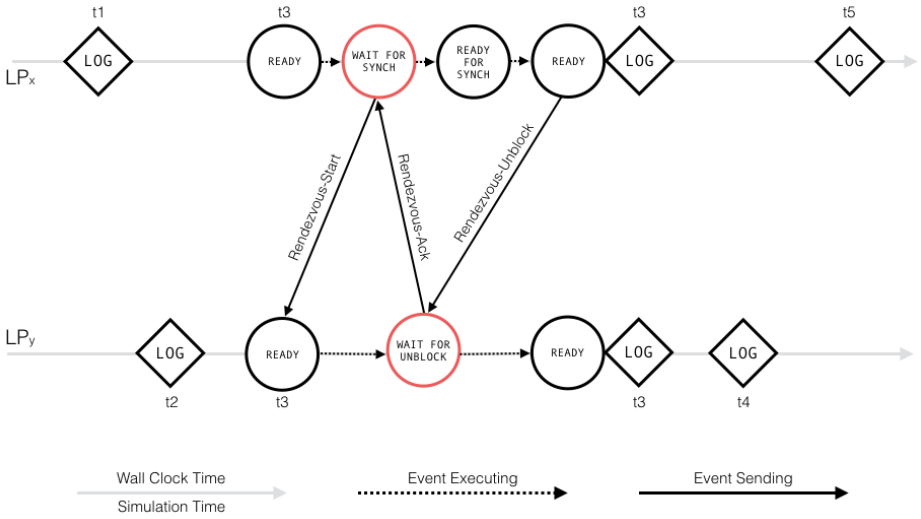

Figure 5.15: Logs position regards to ECS

simulation objects may receive a straggler event are two (See Figure 5.15) :

1. meanwhile the hitter is in `WAIT_FOR_SYNCH`
2. meanwhile the stricken is in `WAIT_FOR_UNBLOCK`

In the first case LP_x has inside its anti-event queue the anti-event ae^{rs} related to `RENDEZVOUS_START` and therefore during its rollback LP_x simply sends ae^{rs} to LP_y. In the second case from the point of view of LP_y, it exploits a queue, say `rendezvous-queue`, that is orthogonal to the anti-event one in which it stores each `RENDEZVOUS_START` that it has processed: in case of rollback it scans the `rendezvous-queue` and sends to each involved simulation object the corresponding anti-start-event. Regarding the others control messages they do not require any anti-event because if a simulation object receives a control message with a rendezvous mark different from the one that it is waiting for it simply discards it because it can be sure that it is an old control message.

Prior to the introduction of our synchronisation scheme, before dispatching the new event for the current simulation object the schedule function updats the `bound` of the object that it is dispatching, namely the last event correctly

executed by the current object. From now on this update is possible only in some cases. During the execution a cross-state event e the simulation object x will pass through schedule function at least twice: the first time when x starts executing e and the second, after the resolution of the cross-state dependency, when it is re-dispatched with the information for "opening" the required stocks. If at each call of schedule we update the `bound`, we do not correctly complete the simulation event that has raised the dependency. Therefore if a LP holds the `READY_FOR_SYNCH` state its bound will be updated only when it will pass through to `READY` state meaning that it has really completed the cross-state dependency.

In order to simplify the interaction of application programmer, we have added a flag that can be utilised at compile-time of our platform for enabling or disabling our new advanced memory manager and the related synchronisation scheme.

This is how our platform is evolved:

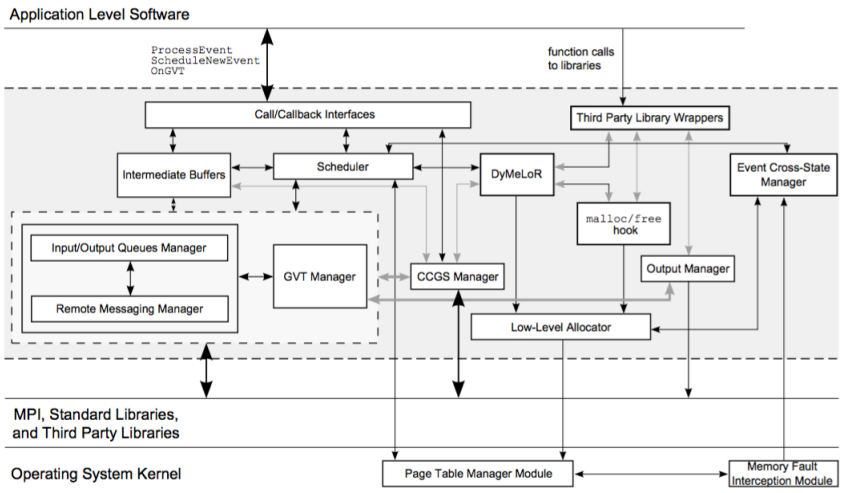

Figure 5.16: New Root-Sim Architecture

5.2.4 Third Party Libraries Handling

Our memory management architecture supports also the third party libraries because it cooperates with the DyMeLoR open source allocator which has the goal of handling memory allocation needs in optimistic PDES platforms. DyMeLoR itself wraps all function needed by ANSI-C stateless libraries and third-party libraries: it handles any memory allocation demanded from these libraries (e.g. `strdup`). Therefore, in case of simulation objects call one of these libraries no updates are required in terms of paging data-structures while they are running in simulation-object mode, otherwise such updates may be materialised by the dynamic linker over the SIBLING page table rather than on the ANCESTOR one. Except for the `malloc` and `stdio`, no stateful libraries (e.g. strtok) are supported yet because the data-structures needed by these for handling read and write operation triggered by whatever concurrent simulation events may cause indirect cross-state dependencies that may hamper the performances of our advanced memory management. The work in [52] shows how to provide consistency for I/O operations in case of optimistic environment in a transparent manner for the application code. Offering support for stateful libraries will be the goal of future works.

CHAPTER 6

Experimental Evaluation

In this chapter we describe the simulation model that we have used for our experimental as well as we discuss our experimental results.

6.1 Model

We have implemented the support for ECS within the open source ROme OpTimistic Simulator (ROOT-Sim)[2].

In this section we provide experimental data achieved by testing our proposal running the implementation of a multi-robot exploration and mapping simulation model, according to the results in [53]. 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 repre-

sentation 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:

1. they use their sensors to estimate their mutual physical position, recall that they are just in *proximity*
2. they verify the goodness of their position hypothesis by creating a rendezvous point (not to be confused with rende-vous control messages in our synchronisation protocol) in the explored part of the region, and trying to meet again there
3. 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 2 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 [53]) 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, estimating the respective position of the agents, 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 [54]. 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 testing the innovative programming paradigm based on cross-state access and consequent adoption over NUMA. In our implementation, we rely on two different types of LPs, namely active ones (implementing the robots) and passive ones (implementing regions of the exploration 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 agents, and at the same time allows us to define proximity regions which are used by the agents to detect the presence of other robots 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 agents 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.

6.2 Tests

To test the ECS proposal with and without NUMA facilities, we have compared the execution time for this simulation model when run without ECS therefore 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, then with ECS enable and in the end with both ECS and NUMA facilities enable. 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 6.1 for

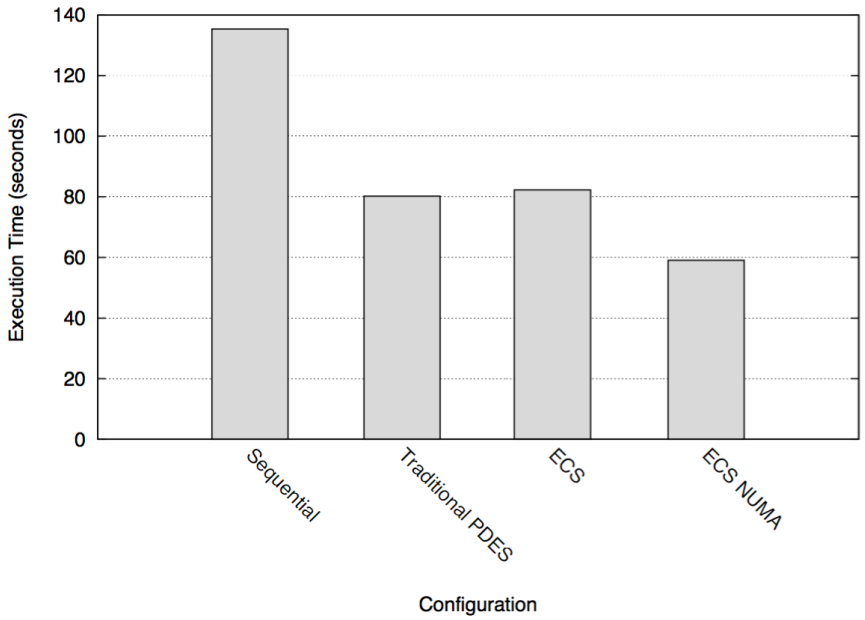

Figure 6.1: Experimental Results

the different settings of the underlying simulation engine (where each reported sample is averaged over 10 runs).

By the results we observe that ECS protocol is a little bit more expensive than traditional PDES coding paradigm. Therefore we can affirm that we have allow application-level programmer to code according to a sequential-style inside a parallel environment without lost performances. The noteworthy result is achieved activating the NUMA supports, in this way ECS provides high performance: the execution time becomes 2 times lower compared to both the traditional PDES case and the ECS without NUMA facilities. This improvement is caused by the fact that now thanks to the ad-hoc `ECS_allocator`, the `NUMA_allocator` can move the entire memory area related to the state of each LP toward the must suitable NUMA node, while previously it was able to migrate only general page segments. In this way any NUMA node can address with low latency the entire state of each simulation object, handled by one of its CPU-cores.

CHAPTER 7

Conclusion

ECS (Event and Cross-State): the new protocol for synchronizing the execution of concurrent simulation objects forming a DES model is the answer to the Fujimoto question[1] in the context of shared-memory multi-core NUMA architecture. We have provided a more general programming and execution model than the traditional PDES. The state portion that can be now accessed by each LP during its execution is not limited to its own state or to the shared global variables only. Now, an LP is allowed to access the state of whichever simulation objects both in read and write mode . Each simulation object has the possibility of accessing the others LPs' states as in a sequential-style DES execution, where the latter pass to the former a pointer to their states inside the payload of simulation message. We have created an advanced memory management architecture together with an advanced synchronisation mechanism that can guarantee consistency and progress. Our work supports cross-state access, joints to concurrency and speculative processing, in an application transparent manner. All our work has been done targeting NUMA machines.

The results have proofed what we claimed: it is possible augmenting a PDES environment with capabilities that allow application-level programmer to rely sequential-style coding approach, where any memory location is implicitly ac-

cessible while processing any simulation event.

Using together with ECS protocol the facilities of `NUMA_allocator`, better performance be achieved since now it is possible to move the entire memory area related to the state of each LP toward the must suitable NUMA node proving low latency memory accesses.

Thanks to our new protocol, a better performance can be achieved. During the past the only way to share big amount of data inside a PDES platform was cross-scheduling of events, therefore sending large data entailed huge overhead due to the copy of the entire interested memory area. Now, thanks to our solution these expensive copies are no longer required, an LP can simply send to another simulation object its pointer to the real data enveloped inside a simulation message.

However, our solution entails huge overhead due to many rollbacks for aligning the involved simulation object to the required simulation time. The natural evolution of our protocol is the clusterisation of simulation object. Instead of scheduling single LP, our environment could take into account sets of LPs, where each set is composed by all LPs that are linked together by multiple/repeated event cross-state dependences. In this way all LPs that show affinity are grouped and therefore they must not be synchronised for granting access the state of each other. They evolve all together like in a sequential environment. This assumption is guaranteed if the scheduler still dispatches according to the STF algorithm. Groups require a new synchronisation protocol to agree upon the start-time of group execution, a new scheduling scheme and a consequent new rollback police as well as a redesigned coasting-forward phase. All these aspects have been targeted by Nazzareno Marziale in [55].

Bibliography

[1] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, October 1990.

[2] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. The rome optimistic simulator: Core internals and programming model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, pages 96–98, ICST, Brussels, Belgium, Belgium, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[3] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.

[4] Wieland F. and Jefferson D.R. Case studies in serial and parallel simulation. In *In Proceedings of the 1989 International Conference on Parallel Processing*, volume 3, pages 255–258, August 1989.

[5] K Mani Chandy and Jayadev Misra. Distributed simulation: A case study in design and verification of distributed programs. *Software Engineering, IEEE Transactions on*, (5):440–452, 1979.

[6] MPI Forum. Message Passing Interface Forum. http://www.mpi-forum.org/, 1994.

[7] R. M. Fujimoto. The virtual time machine. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, pages 199–208, New York, NY, USA, 1989. ACM.

[8] Richard M Fujimoto. Parallel and distributed discrete event simulation: algorithms and applications. In *Proceedings of the 25th conference on Winter simulation*, pages 106–114. ACM, 1993.

[9] Paul F Reynolds Jr. A spectrum of options for parallel simulation. In *Proceedings of the 20th conference on Winter simulation*, pages 325–332. ACM, 1988.

[10] Alois Ferscha and Satish K Tripathi. Parallel and distributed simulation of discrete event systems. 1998.

[11] K. Mani Chandy and Jayadev Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–206, 1981.

[12] Randal E Bryant. A switch-level model and simulator for mos digital systems. *Computers, IEEE Transactions on*, 100(2):160–177, 1984.

[13] Richard Fujimoto and David Nicol. State of the art in parallel simulation. In *Proceedings of the 24th conference on Winter simulation*, pages 246–254. ACM, 1992.

[14] David M Nicol. Principles of conservative parallel simulation. In *Proceedings of the 28th conference on Winter simulation*, pages 128–135. IEEE Computer Society, 1996.

[15] David M Nicol. *Parallel discrete-event simulation of FCFS stochastic queueing networks*, volume 23. ACM, 1988.

[16] Steven Bellenot et al. Global virtual time algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 22, pages 122–127. Society For Computer Simulation, San Diego, CA, 1990.

[17] Yi-Bing Lin and Edward D Lazowska. Determining the global virtual time in a distributed simulation. In *ICPP (3)*, pages 201–209, 1990.

[18] Christopher D Carothers, Kalyan S Perumalla, and Richard M Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 9(3):224–253, 1999.

[19] David Jefferson. Virtual time ii: storage management in conservative and optimistic systems. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 75–89. ACM, 1990.

[20] Alessandro Pellegrini, Roberto Vitali, Sebastiano Peluso, and Francesco Quaglia. Transparent and efficient shared-state management for optimistic simulations on multi-core machines. In *Proceedings 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 134–141. IEEE Computer Society, August 2012.

[21] Yi-Bing Lin and Edward D Lazowska. *Reducing the state saving overhead for Time Warp parallel simulation.* University of Washington, Department of Computer Science, 1990.

[22] Steven Bellenot. State skipping performance with the time warp operating system. In *6th Workshop on Parallel and Distributed Simulation*, volume 24, pages 53–64, 1992.

[23] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[24] Avinash C Palaniswamy and Philip A Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *ACM SIGSIM Simulation Digest*, volume 23, pages 127–134. ACM, 1993.

[25] Robert Rönngren and Rassul Ayani. Adaptive checkpointing in time warp. In *ACM SIGSIM Simulation Digest*, volume 24, pages 110–117. ACM, 1994.

[26] Sven Sköld and Robert Rönngren. Event sensitive state saving in time warp parallel discrete event simulations. In *Proceedings of the 28th conference on Winter simulation*, pages 653–660. IEEE Computer Society, 1996.

[27] Hassan Rajaei, Rassul Ayani, and Lars-Erik Thorelli. The local time warp approach to parallel simulation. In *ACM SIGSIM Simulation Digest*, volume 23, pages 119–126. ACM, 1993.

[28] Richard M Fujimoto. *Parallel and distributed simulation systems*, volume 300. Wiley New York, 2000.

[29] Roberto Toccaceli and Francesco Quaglia. Dymelor: Dynamic memory logger and restorer library for optimistic simulation objects with generic memory layout. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 163–172. IEEE Computer Society, 2008.

[30] Quaglia F. Pellegrini A., Peluso S. and Vitali R. Transparent speculative parallelization of discrete event simulation applications using global variables.

[31] Cristina Montañola-Sales, Joan-Francesco Gilabert-Navarro, Josep Casanovas-Garcia, Clara Prats Soler, Daniel López Codina, Joaquim Ribas Valls, Pere Joan Cardona Iglesias, and Cristina Vilaplana. Modeling tuberculosis in Barcelona. A solution to speed-up agent-based simulations. In *Proceedings of the 2015 Winter Simulation Conference*, pages 1295—-1306. IEEE Computer Society, 2015.

[32] Pierangelo Di Sanzo, Francesco Quaglia, Bruno Ciciani, Alessandro Pellegrini, Diego Didona, Paolo Romano, Roberto Palmieri, and Sebastiano Peluso. A Flexible Framework for Accurate Simulation of Cloud In-Memory Data Stores. *Simulation Modelling Practice and Theory*, 2015.

[33] Alessandro Pellegrini and Francesco Quaglia. Numa time warp. In *Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation*, pages 59–70. ACM, 2015.

[34] Douglas W. Jones. Concurrent simulation: An alternative to distributed simulation. In *Proceedings of the 18th Conference on Winter Simulation*, WSC '86, pages 417–423, New York, NY, USA, 1986. ACM.

[35] D. W. Jones, C.-C. Chou, D. Renk, and S. C. Bruell. Experience with concurrent simulation. In *Proceedings of the 21st Conference on Winter Simulation*, WSC '89, pages 756–764, New York, NY, USA, 1989. ACM.

[36] David Bruce. The treatment of state in optimistic systems. In *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation*, PADS '95, pages 40–49, Washington, DC, USA, 1995. IEEE Computer Society.

[37] Alessandro Fabbri and Lorenzo Donatiello. Sqtw: A mechanism for state-dependent parallel simulation. description and experimental study. In *Proceedings of the Eleventh Workshop on Parallel and Distributed Simulation*, PADS '97, pages 82–89, Washington, DC, USA, 1997. IEEE Computer Society.

[38] Boon Ping Gan, Malcolm Yoke Hean Low, Junhu Wei, Xiaoguang Wang, Stephen John Turner, and Wentong Cai. Distributed simulation and manufacturing: Synchronization and management of shared state in hla-based distributed simulation. In *Proceedings of the 35th Conference on Winter Simulation: Driving Innovation*, WSC '03, pages 847–854. Winter Simulation Conference, 2003.

[39] Malcolm Yoke Hean Low, Boon Ping Gan, Junhu Wei, Xiaoguang Wang, Stephen John Turner, and Wentong Cai. Shared state synchronization for hla-based distributed simulation. Technical report, 2006.

[40] Malcolm Yoke Hean Low, Boon Ping Gan, Junhu Wei, Xiaoguang Wang, Stephen John Turner, and Wentong Cai. Shared state synchronization for hla-based distributed simulation. *Simulation*, 82(8):511–521, August 2006.

[41] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, December 2006.

[42] Alessandro Pellegrini. Hijacker: Efficient static software instrumentation with applications in high performance computing (poster paper). In *Proceedings of the 2013 International Conference on High Performance Computing & Simulation*, pages 650–655. IEEE Computer Society, 2013.

[43] Li-li Chen, Ya-shuai Lu, Yi-ping Yao, Shao-liang Peng, and Ling-da Wu. A well-balanced time warp system on multi-core environments. In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, PADS '11, pages 1–9, Washington, DC, USA, 2011. IEEE Computer Society.

[44] K. Ghostand and R.M. Fujimoto. Parallel discrete event simulation using space-time memory. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 201–208, August 1991.

[45] K.M. Chandy, R. Sherman, and University of Southern California. Information Sciences Institute. *Space-time and simulation.* Number No. 238 in ISI reprint series. University of Southern California, Information Sciences Institute, 1989.

[46] Brian W Kernighan and Dennis M Ritchie. *The {C} Programming Language.* Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[47] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Number 253669-033US. December 2009.

[48] R. Vitali, A. Pellegrini, and F. Quaglia. A load-sharing architecture for high performance optimistic simulations on multi-core machines. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10, Dec 2012.

[49] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Principles of Advanced and Distributed Simulation (PADS), 2012 ACM/IEEE/SCS 26th Workshop on*, pages 211–220, July 2012.

[50] D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev. Optimization of parallel discrete event simulator for multi-core systems. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 520–531, May 2012.

[51] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. Di-dymelor: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 45–53. IEEE Computer Society, 2009.

[52] Francesco Antonacci, Alessandro Pellegrini, and Francesco Quaglia. Consistent and efficient output-streams management in optimistic simulation platforms. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, pages 315–326. ACM, 2013.

[53] Dieter Fox, Jonathan Ko, Kurt Konolige, Benson Limketkai, Dirk Schulz, and Benjamin Stewart. Distributed Multirobot Exploration and Mapping. *Proceedings of the IEEE*, 94(7):1325–1339, 2006.

[54] H Mehl. A deterministic tie-breaking scheme for sequential and distributed simulation. In *Proceedings of the Workshop on Parallel and Distributed Simulation*. ACM, 1992.

[55] Nazzareno Marziale. Dynamic clustering of simulation objects in speculative parallel simulation systems. Master's thesis, Sapienza, University of Rome, 1 2016.

Acknowledgements

First of all, I wish to thank *Professor Quaglia*, for believing in us even though the situation prevented us from hoping for an auspicious ending.

I cannot forget the tireless *Alessandro Pellegrini*, without his precious help completing this project would have been unworkable. He has been close to us for the entire project. He has made feasible what seemed unfeasible. He led us to a different way of thinking, continuously reasoning and discussing with us.

I wish to thank the irreplaceable friend, supporter and colleague *Nazzareno*, this project would have been impossible without him.

Billions of thanks to *my mother*, *my father* and *my sister* , for all the sacrifices that they have faced for me. I hope to have made them proud of me, and if so, to continue in this way. I apologize for the constant irritability of the last months.

Thousands of thanks to my second family: *Pina*, *Tonino* and *Elisa* that have embraced me as a son, supporting me along the entire duration of my academic career and not only.

A lot of thanks to *Silvia*, for having tolerated Nazzareno, me and all our stupid jokes.

Special thanks go to my Syrian brother *Obaida*, it has been a pleasure and honour for me meeting him.

Thanks to my cousin *Edoardo* for helping me to never lose my smile.

The list of friends that I want to thank for supporting me is endless. *Marco*, *Giammarco*, *Ilaria*, *Antonio*, *Lele*, *Alessio*, *Giulio*, *Francesca Romana*, *Katarina* and so on. Without you this path would have been hard.

I cannot forget my special friends of "*Società Źubrówka*", that made the last two years amazing.

I cannot forget *Marilyn* and her efforts to teach me English.

Last but not least *Stephanie*. She has supported me giving the energy and courage to always go ahead. She has helped me to deeply analyse all situations in order to find the best solution. I would have not been able to get at this stage without her.