SAPIENZA
UNIVERSITÀ DI ROMA

FACULTY OF ENGINEERING

Master's thesis in

MASTER OF SCIENCE IN ENGINEERING IN COMPUTER SCIENCE

Transactional Memory Based Speculative Parallel Execution of Discrete Event Applications

Candidate
Mauro Ianni

Thesis Advisor
Prof. Francesco Quaglia

Co-Advisor
Eng. Alessandro Pellegrini

Academic Year 2014/2015

So long, and thanks for all the fish.

Contents

1 Introduction **3**

2 Discrete Event Simulation **10**

2.1 Formal Definition of DES Models 11

2.2 Systemic approach to DES 19

2.2.1 Basic Components of a Discrete-Event Simulation . . . 22

2.2.2 Simulation Kernel's Basic Logic 26

2.3 Parallel Discrete Event Simulation 28

2.3.1 Synchronization Problem 33

2.3.1.1 Conservative Approach 34

3 Speculative PDES **39**

3.1 Recoverability 40

3.1.1 Rollback Supports 46

3.1.2 Additional Components of Speculative PDES 51

3.2 Software Solutions 55

3.2.1 State Saving & Restore 55

3.2.1.1 Copy State Saving (CSS) 56

3.2.1.2 Sparse State Saving (SSS) 57

3.2.1.3 incremental State Saving (ISS) 59

4 Moving to Hardware Based Recoverability **62**

4.1 Framing the HTM based proposal within literature 65

4.2 Hardware Transactional Memory (HTM) 69

4.3 HTM-based Speculative PDES 72

4.3.1 Structures 73

4.3.2 Basic Engine 76

4.3.3 Optimizations 83

4.3.3.1 Handling Simultaneous Events 83

4.3.3.2 Non-zero Lookahead 84

4.3.3.3 Throttling 87

4.4 HTM-based vs

Classical Speculative PDES: a Summary 89

5 Experimental evaluation **92**

5.1 Hardware Setup 92

5.2 Benchmark Applications 93

5.3 Results 94

6 Conclusions **102**

7 Bibliography **104**

Abstract

Speculative parallel processing is a well known means to deliver high performance and scalability when executing discrete event simulation models. Nevertheless, it requires the runtime support to restore the application's state to some past (consistent) image. Traditionally, the recoverability support has been realized via proper software layers. However, although a lot of optimizations have been provided in literature for making software-based recoverability highly efficient, its relative overhead may still represent an impairment to performance in case of (very) fine grain applications. This work presents an innovative runtime support for speculative parallel processing of discrete event simulation models on multi-core architectures, which exploits Hardware-Transactional-Memory (HTM) facilities, nowadays offered by off-the-shelf processors, for the purpose of state recoverability. In this thesis, the speculative updates on the state of the simulation model are executed as concurrent HTM-based transactions that are also in charge of detecting whether the update is consistent with the advancement of logical-time along model execution. This is achieved by including in the HTM-based transactional code-block both the activation of the application layer in charge of processing the simulation event, and the execution of housekeeping tasks aimed at determining the safety (in terms of causal consistency) of the executed transaction. This proposal is fully transparent to the application code. Hence,

this HTM-based run-time support can host conventionally developed discrete event models relying on the concept of event-handlers to be dispatched by an underlying simulation engine. Experimental data show that this proposal provides 75% to 92% of the ideal speedup on an Intel Haswell based platform (equipped with 4 physical cores and HTM support) for discrete event models with event granularity ranging between 2 and 12 microseconds. The data also show that these same models cannot be executed efficiently on top of a last generation parallel discrete event simulation platform employing software-based recoverability.

Chapter 1

Introduction

The recent trend in computing architectures has shown that multi-/many-core machines have become the reference platforms to provide a continuous increase of computing power [1]. In the context of Parallel Discrete Event Simulation (PDES) [2], speculative execution of simulation models, incarnated by the Time Warp synchronization protocol [3] (or variants of it), has already been proven to be able to exploit the scaled-up computing power offered by such platforms (see, e.g., [4]) in order to speedup the execution of very large and complex models.

In Parallel Discrete Event Simulation (PDES) [2], the simulation model is partitioned into simulation objects, historically referred to as Logical Processes (LPs), that are allowed to be dispatched for event processing along concurrent worker-threads. This allows for exploiting hardware parallelism with the aim at speeding up model execution. The simulation object is usually implemented as a set of data structures to be updated via a call-back (representing the application entry point), which is dispatched by the underlying PDES platform (see, e.g., [5, 6, 7]). The dispatch operation corresponds to the processing of a timestamped event at the simulation object,

and causally consistent execution is typically based on forcing any simulation object to process its input events in non-decreasing timestamp order, including those produced by other objects as the result of processing activities they carried out. In fact, although recent approaches provide alternative object-interaction methods (see, e.g., [8, 9]), the cross-scheduling of events across simulation-objects is the basic approach adopted in PDES in order to model the interactions occurring between the entities belonging to the simulated system/scenario.

In speculative PDES [3] there is no preliminary assessment of causal consistency of the events, rather they are dispatched for execution on whichever simulation object as soon as they are available. This leads to high exploitation of the intrinsic parallelism in the simulation model, since causally unrelated portions of the simulated state trajectory can be processed with no a-priory synchronization of the execution of the different simulation objects. However, if the updates occurring along a computation path are eventually detected to be inconsistent (i.e. they occurred out of timestamp order), rollback mechanisms need to be actuated so as to restore the application state to a consistent (past) snapshot from which forward computation can be resumed.

Although simple in principle, state recoverability of the simulation objects poses problems on the side of both performance and application transparency. In fact, the more efficient the recoverability support, the lower its overhead. On the other hand, application-transparent state restore typically demands more operations from an underlying recoverability layer, which further biases the tradeoff away from pure performance optimization. Literature studies have (jointly) addressed performance and transparency aspects in state recoverability of simulation objects via disparate checkpointing techniques

[10, 11] that, except for a few proposals based on (either conventional or non-conventional) hardware support [12, 13], rely on software implementations of the checkpointing support. Although most of these proposals also entail overhead minimization techniques (e.g. via tuning of the parameters driving both checkpointing and–consequently–state recovery operations), for the case of very fine grain simulation models, namely models based on events that require a few microseconds of CPU-time for being processed, the overhead can still represent an impairment to performance. A way to cope with this issue is the alternative recoverability technique based on reverse computing [14], where the forward application code is coupled with a (in some cases automatically generated [15]) reverse code version that is used to undo the state updates that are eventually revealed to be inconsistent. This solution pays off especially in contexts where, beside having fine grain reverse (hence forward) events, the portion of the state trajectory to be undone (namely the so called *rollback length*) is short, which leads to a reduced number of reverse events to be processed per rollback operation.

Another aspect that plays a relevant role in case of speculative PDES with very fine grain models is the cost associated with cross-simulation-object scheduling of events, which may become predominant. This is typically achieved via message exchange (managed at the level of the underlying PDES platform), and the classical approach to undo the notification of an event that has been scheduled as a result of the processing of another event that is then detected to be non-consistent is to send a negative copy of it (the so called *anti-message*). Beyond potentially triggering a rollback operation at the recipient (in case the original copy of the message—namely event—was already processed) anti-messages lead to doubling the communication cost per-incorrect-scheduled events. To cope with this issue, literature

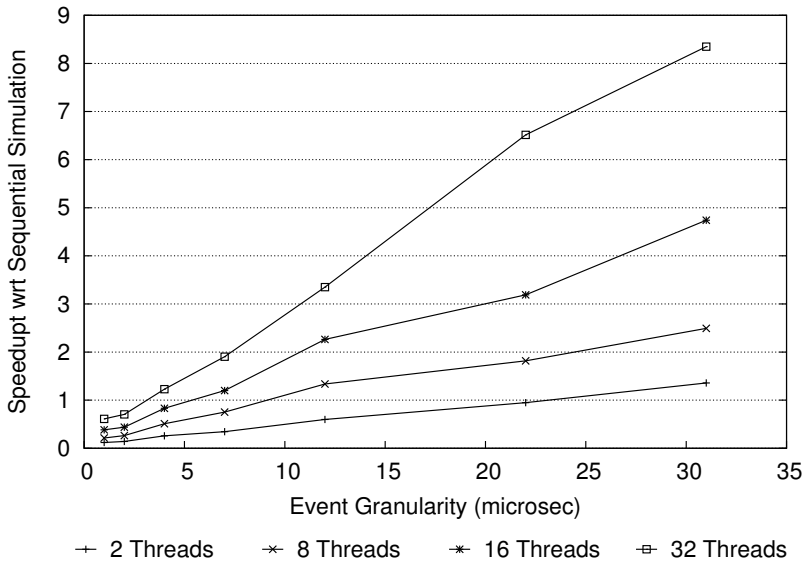

Fig. 1.1: Speedup for PHOLD while varying event granularity and number of threads

approaches have been proposed in order to reduce the number of message exchange operations, such as the ones based on message aggregation [16] or lazy-cancellation (lazy-antimessages) [17].

In any case, despite the existence of a bunch of literature results on optimizing speculative PDES systems, executing simulation models with very fine grain events on top of these systems in a performance-efficient manner is still a non-trivial achievement. Just to provide some empirical evidence, is reported in Figure 1.1 the speedup achievable by running the classical PHOLD benchmark for PDES systems [18] (in a configuration with 2048 simulation objects) on top of the ROOT-Sim last generation speculative PDES platform ([1]) hosted on a 32-core off-the-shelf HP ProLiant machine, with respect to the sequential simulation of the same benchmark (same code) on a calendar-queue scheduler (still executed on the same machine). In the plot,

[1]https://github.com/HPDCS/ROOT-Sim

the CPU-demand by PHOLD events is varied from a few to some tens of microseconds. The plotted curves show that speedup is unacceptable (it is a slow-down) for minimal CPU-requirements by the events, and is anyhow non-competitive (vs the employed number of threads) even when events last tens of microseconds.

In this thesis I cope with the issue of speculatively running PDES applications with (very) fine grain PDES applications efficiently on top of multi-core machines, which is achieved by exploiting the Hardware-Transactional-Memory (HTM) support that is nowadays offered by off-the-shelf processors (such as the Intel Haswell). Overall, my proposal is suited for contexts where conventional speculative PDES engines based on software recoverability (even the most advanced ones) fail to provide speedup just due to the excessively fine granularity of the simulation events (as we have shown in Figure 4.1).

With this proposal we speculatively execute an event as an HTM-based transaction that includes the actual buffering of any newly produced event destined to whichever simulation object (in case the transaction is successfully committed), and which entails a code-block that is used to explicitly detect whether the transaction (hence the processed event) is causally consistent, so that a commit is issued only in case consistency is verified. On the other hand, if the transaction is not guaranteed to be causally consistent, it simply issues an abort command that allows: (A) automatically undoing the updates issued on the state of the simulation object and (B) automatically discarding any new event produced as a result of incorrect event processing. Both these targets are achieved with no intervention by any software layer thanks to the fact that the HTM transactional cache keeping the state updates and the newly produced events is simply squashed upon the abort of the transaction. A secondary effect by our approach is that we allow

intra-simulation-object concurrency in the speculative processing scheme, as opposed to traditional PDES engines where a single event at a time can be CPU-scheduled for a specific simulation object. Specifically, in our approach two worker-threads operating within the PDES environment are allowed to concurrently execute two different events targeting the same simulation object (just depending on how the overall set of events destined to the different simulation objects is clustered along the simulation time axis). If the two events are actually independent (given that their read/write sets are disjoint, which means that they touch different portions of the simulation object state) then they are both committable in our execution model. This leads our system to implement the so called weak-causality model [19], just in the form of parallelization of the execution of events within the same simulation object. As a last note, this approach is application transparent given that the application layer can still be designed as a set of event-handlers touching application specific data structures, as it commonly occurs in reference speculative PDES environments (see [5, 6, 7]) not relying on HTM facilities and sequential simulators as well.

This thesis also reports the results of an experimental study based on running our system, and specific test-bed applications, on top of a machine equipped with 2 quad-core (hyper-thread) Intel(R) Xeon(R) 3.5 GHz processors (with HTM support) and 24 GB RAM, running Linux Ubuntu 12.04.2 LTS, kernel version 3.5.0-23-generic. By the data, the presented HTM-based simulation engine allows achieving 75% to 92% of the ideal speedup and a performance gain of up to 10x vs the last generation ROOT-Sim speculative PDES platform relying on software-based recoverability support. Overall, the approach presented in this thesis opens the possibility to efficiently parallelize the speculative execution of discrete event simulation models that are

de-facto not efficiently parallelizable by relying on state of the art speculation and software-based recoverability techniques.

Chapter 2

Discrete Event Simulation

This thesis is framed within the context of *simulation*, namely the imitation of the evolution of a real system or process along time. A first step to simulate something requires the development of a model, that represents the main characteristics of the functions/behaviours of the simulated environment. The model represents the system, meanwhile the simulation represents the execution over time. The simulation is used to predict the effects of alternatives conditions and actions [20].

The Event Based Simulation is a simulation methodology where, to describe the execution, a sequence of events distributed over time is exploited. Each event is located in time and brings changes in the state of the system. In particular, this work is focused in *Discrete Event Simulation* (DES). An event is considered discrete because it is characterized by an impulsive duration, then it begins at a time instant and it is considered to end in the same instant (the begin coincides with the end). Between two consecutive events, no changes in the systems occur, it is therefore possible to jump from one event to the next. This means that the execution, during the simulation, advances according to the timestamp of each event.

This approach differs from *continuous simulation* in which the execution is characterized by events with a duration, then in each instant of time (small time slices) it has to consider the set of activities happening concurrently. Since discrete-event simulations have not to simulate every time slice, they are typically faster.

There are two principal approaches for DES, one based on a *formalism* to model and analyze discrete-event systems, and another one based on a set of *methodologies and tools* to support the execution of a simulation model. Both the approaches will be presented, underlining weaknesses and strenghts [21]. In the end the original sequential approach will be extended with a parallel version built to run on shared memory multi-/many-cores architectures, so called Parellel Discrete Event Simulation (PDES), which will be considered in this thesis with the aim of improving it.

2.1 Formal Definition of DES Models

It is possible to rely on *Discrete Event Systems Specification* (DEVS) to give a formal definition of simulations model [22, 23]. DEVS, introduced for the first time in the 1976 [24], is a hierarchical and modular formalism to analyze and model general systems, that represents an extension of the formalism of the Moore machine [25]. In the Moore machine formalism we have a finite state automaton the output of which is determined only by its state. Differently, DEVS output depends directly from the input. Moreover DEVS assigns a duration to each state and provides hierarchical concept with an operation, called *coupling*.

Essentially, DEVS provides a formalism utilized either to have a general understanding of discrete-event systems and to design hierarchically decom-

posable discrete event models, uncoupling them from the models generated by the computer. At the same time offers a framework to generate models and execute them by its abstract simulator concepts. In the classic DEVS formalism, *coupled DEVS* describes the structure of the system, while *atomic DEVS* captures its behaviour.

The formalism defines the *atomic model* as a 8-tuple:

$$M = \langle X, S, s_0, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{2.1}$$

Where

- X is the set of external input events;
- S is the set of sequential states (called *set of partial states*). The definition of the state can be extended with the state variable σ, which represents the maximum time spent in a state before triggering an internal transition, if no external events are received;
- s_0 is the initial state of the simulation;
- Y is the set of output events;
- $\delta_{int} : S \to S$ is the internal transition function which defines how a state of the system changes internally (to which state the system transits), when a lifetime period is passed without external events arrived;
- $\delta_{ext} : Q \times X \to S$ is the external transition function which defines how the arrive of an external input event changes the state of the system, where $Q = \{(s, e)| \in S, 0 \leq e \leq ta(s)\}$, and e is the elapsed time since the last event;

- $\lambda : S \rightarrow Y^{\phi}$ is the output function that defines how a state of the system generates an output event when the elapsed time reaches the lifetime of the state;

- $ta : S \rightarrow R_{0 \rightarrow \infty}$ is the time advance function, which is used to determine the lifespan of a state; when the time assigned to the event is finished, a trigger starts a new transition; when σ is present, its value is returned.

A model designed following the DEVS formalism, according to the Equation 2.1, transits along the states in S following its transition functions. When no events come, the time of the model progresses according to the function ta that is applied to the current state. Starting by the old state, a new state is determined by the function δ_{int}. The output events are generated by the system before an internal transition takes place. If an external event occurs, the function δ_{ext} produces a state transition, starting by the old state, based on the external event itself and the time spent in the old state.

As mentioned before, the structure of the system is defined by the *coupled model* as follow:

$$DN = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle \tag{2.2}$$

Where

- X_{self} is the set of external input events handled by the coupled model;

- Y_{self} is the set of output events handled by the coupled model;

- D is the name set of sub-components, i.e the *name set*;

- $\{M_i\}$ is the set of sub-components such that $\forall i \in D : M_i$ can be either an atomic or a coupled DEVS model, i.e. a model definition;

- $\{I_i\}$ is a set such that $\forall i \in D \cup \{self\} : I_i$ are the *influences* of i, i.e. the external input coupling;

- $\{Z_{i,j}\}$ is a set such that $\forall j \in I_j : Z_{i,j}$ is the translation of the ouput from i to j, i.e. a function that maps the events generated by a model in D to another model in D;

- $select : 2^D \rightarrow D$ is the *tie-breaker function*, which defines the way to select one event from all the simultaneous events.

In the end, the just defined structure is subject to the constraints that, for $\forall i \in D$, the corresponding model is defined according the Equation (2.1), like:

$$M_i = \langle X^i, S^i, s_0^i, Y^i, \delta_{int}^i, \delta_{ext}^i, \lambda^i, ta^i \rangle \tag{2.3}$$

and that:

- $I_i \subseteq D \cup \{self\}, i \notin I_i$, i.e. the influences of i have to come from the available model definition, and i can not be influenced by itself

- $Z_{self,j} : X_{self} \rightarrow X_j$, i.e. input events handled by the coupled DEVS model can be mapped to input event for the model $j \in D$;

- $Z_{i,self} : Y_i \rightarrow Y_{self}$, i.e. output events that are generated by any sub-component structure $i \in D$ can be managed by the coupled DEVS model;

- $Z_{i,j} : Y_i \rightarrow X_j$, i.e. output events generated by any sub-components structure $i \in D$ can be mapped into external input events for any sub-component model $j \in D$.

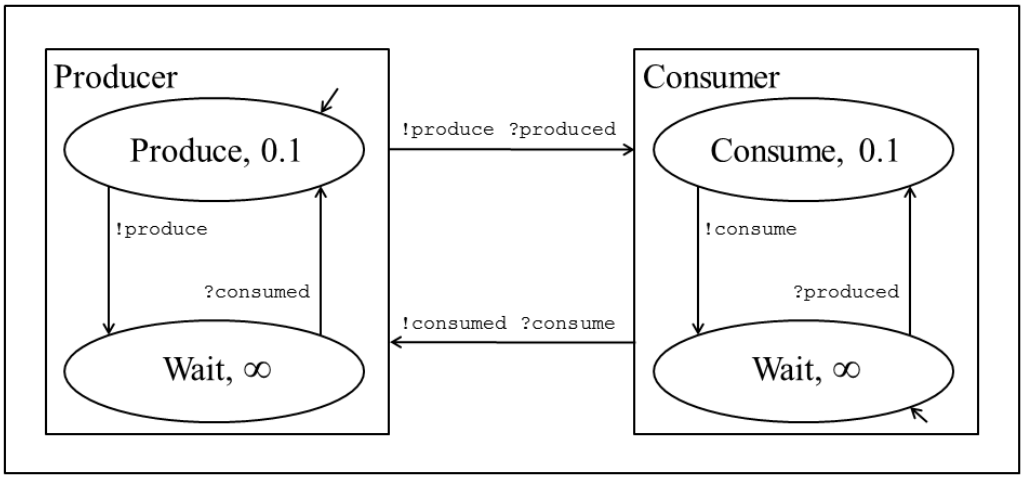

Fig. 2.1: Sample DEVS model: Producer/Consumer

Then, a coupled DEVS model, defines how to connect the different subcomponents of the model together to generate a new model. The new generated model is a DEVS model itself (thanks to the closure property under coupling [22]), and then can be a new component to employable in a bigger coupled model. This means that the components structured in $\{M_i\}$ can be defined according to both Equation 2.1 and Equation 2.2. This is the *hierarchical* aspect of model composability mentioned at the begin.

Now it is possible to provide an example to deeply understand the concepts expressed so far. The diagram in Figure 2.1 represents a DEVS model description of a consumer-producer system. The behaviour of the system is described by external and output events. In this example the external events are `?consumed` and `?produced`, and the output events are `!consume` and `!produce`. The consumer has the states *Consume* and *Wait*, and starts the simulation in the second one. The producer has the states *Produce* and *Wait*, and starts the simulation in the first one. Both *Produce* and *Consume* take 0.1 seconds to respectively, produce an item, for the producer, and consume

it for the consumer. Starting by the *Produce* state of the producer, when 0.1 seconds have passed, an item is produced, and then the output event `!produce` is triggered. In the same way, starting from the *Consume* state of the cosumer, when 0.1 seconds have passed, the item is consumed and the output event `!consume` is triggered.

The aim of this simulation is to connect two models (the consumer and the producer) to create a bigger simulation model. First of all we have to formalize the individual *atomic* DEVS models for the producer and the consumer, following the formalism expressed in Equation 2.1:

$$Producer = \langle X^P, S^P, s_0^P, Y^P, \delta_{int}^P, \delta_{ext}^P, \lambda^P, ta^P \rangle \tag{2.4}$$

where

$X^P = \{\texttt{?consumed}\}$

$Y^P = \{\texttt{!produced}\}$

$S^P = \{(d, \sigma) | d \in \{Produce, Wait\}, \sigma \in [0, \infty]\}$

$s_0^P = (Produce, 0.1)$

$ta^P(s) = \sigma, \forall s \in S$

$\delta_{ext}^P(((Wait, \sigma), t_e), \texttt{?consumed}) = (Produce, 0.1)$

$\delta_{int}^P(Produce, \sigma), t_e) = (Wait, \infty)$

$\delta_{int}^P(Wait, \sigma), t_e) = (Produce, 0.1)$

$\delta^P(Produce, \sigma) = \texttt{!produce}$

$\delta^P(Wait, \infty) = \emptyset$

and in the same way:

$$Consumer = \langle X^C, S^C, s_0^C, Y^C, \delta_{int}^C, \delta_{ext}^C, \lambda^C, ta^C \rangle \tag{2.5}$$

where

$X^C = \{\texttt{?produced}\}$

$Y^C = \{\texttt{!consumed}\}$

$S^C = \{(d, \sigma) | d \in \{Consume, Wait\}, \sigma \in [0, \infty]\}$

$s_0^C = (Consume, 0.1)$

$ta^C(s) = \sigma, \forall s \in S$

$\delta_{ext}^C(((Wait, \sigma), t_e), \texttt{?produced}) = (Consume, 0.1)$

$\delta_{int}^C(Consume, \sigma), t_e) = (Wait, \infty)$

$\delta_{int}^C(Wait, \sigma), t_e) = (Consume, 0.1)$

$\delta^P(Consume, \sigma) = \texttt{!consume}$

$\delta^P(Wait, \infty) = \emptyset$

The resulting final simulation model, can be then composed merging together the two atomic models expressed by the Equations 2.3 and 2.4 as a new coupled model expressed according Equation 2.2:

$$DN = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle \tag{2.6}$$

where

$X_self = \{\}$

$Y_self = \{\}$

$D = Producer, Cosumer$

$\{M_{Producer}, M_{Consumer}\}$ defined according the equations 2.3 and 2.4

$I_i = \{\}$

$Z_{i,j} = \{(Producer.\texttt{!produce}, Consumer.\texttt{?produced}),$
$(Consumer.\texttt{!consume}, Producer.\texttt{consumed})\}$

The use of this formalism to describe also simple models, introduces two great advantages.

First, this formalism allows to automate the process of *verification* and *validation*, in order to check if it is credible and accurate [26, 27]. This factor is very important because a model is only an imitation of what actually happens in the real world and therefore can not reproduce it in the exact same way. Thus, it should be validated and verified to the extent necessary depending on the purpose of the application. Second, the so defined model can be exported into a computer model, that can be processed by a simulation algorithm. In this way the execution of the model can be automated, with all the advantages that follow.

For the first point, when the DEVS model is an extension like Finite and Deterministic DEVS (FD-DEVS) [28], or Schedule-Preserving DEVS (SP-DEVS) [29], or still Finite and Real-Time DEVS (FRT-DEVS) [30], it has been proved an isomorphic finite structure can be obtained starting from the original infinite structure of the model. In this way a reachability graph can be obtained starting from the structure, allowing, for example, to decide if there are dead-locks or live-locks in the model [28, 30, 31], and, in the case

of SP-DEVS, which are the maximum and the minimum execution bounds in time for the selected model [29].

About the second point, to translate the formal definition of the model in a computer simulation, the atomic and the coupled models are handled by DEVS in different way. The simulation of an atomic model is performed by a *simulator*. The simulation of a coupled model is performed by a *coordinator*. In fact the main role of the coordinator is to manage the simulation of the different atomic models: it enforces *time synchronization*, controlling the advancement of the simulation time in the different atomic models, keeping them always aligned; it supports *message propagation* between the different simulators, transmitting the input and output messages between the associated coupling, defined in the coupled model.

In the end, one of the most important advantages of this formal approach to the discrete event simulation is that, in this way, the simulation model exists regardless of its actual implementations. This allows the analysts to study its properties and produce the simulation model before its implementation, without particular knowledges about the used tool-kit. The analysis of the model can be very difficult, then is better to decouple the formal construction of the model and its translation to a computer simulator.

2.2 Systemic approach to DES

The systemic approach to DES faces the topic of discrete event simulation more pragmatically, focusing much more on the supports for the development on the model and the effective implementation of the software that takes care of simulation, normally called *simulation core* or *simulation kernel*. Also if the attention is focused on different issues, many aspects are taken directly

from the DEVS formal definition of the model. However, the main effort has been focused in the development on the techniques of simulation with the goal to maximize the overall throughput of the execution. In this new scenario a model is composed by:

- the joint union of *simulation states*, where the variable that are used to keep track of the evolution of the studied system are stored;
- the set of events E, that corresponds to the phenomenons of the real world studied and that with their execution modify the state of the simulation model;
- a transition function $\sigma(s, e) : S \times E \rightarrow S$ that, at the time when an event $e \in E$ occurs, determines the transition from the previous simulation state s to the new simulation state s'.

As seen before, each discrete event e corresponds to a time T_e during the simulation run, and the execution of the event itself makes the global simulation time to advance until its time. In fact, during the execution of a discrete event simulation, the *simulation time* (ST) does not have a homogeneous progression, but advances following the sequence of events; this describes a logical time that is associated with the simulation model and can be expressed in different time units according to the type of simulation described by the model (e.g. milliseconds, seconds, years). In this context is very important to distinguish this conceptual notion of time from the *wall clock time* (WCT), that differently describes the time as perceived by the human beings in the real world. Then, the simulation time describes the advancement of the time inside the simulation, meanwhile the wall clock time gives us a feedback about the speed of the simulation executed.

The implementations of DES traditionally used, are directly inherited from the *event driven programming* paradigm. This paradigm is very important in the computer programming field, especially in graphic user interface and other applications that are centred on performing an action as the response to some input. In this paradigm the flow of the application is driven by receiving events (e.g. mouse click), in a way similar to the *interrupts* in operating systems.

This paradigm has proven to be useful in the context of event simulation for their intrinsic affinities: in fact a DES model can be seen as a set of *event handlers* (an asynchronous callback used to pass control to the relative model's code) that capture events, in this case produced by the application itself, and, depending on the type of the event, produce effects on the simulation state. In fact, the set of event handlers are nothing more than the aforementioned transition function $\sigma(s, e)$, that, taken the current state and the received event, produces the new state for the simulation.

At the begin of the simulation, one or more events are inserted to allow the begin of the simulation. Subsequently, during the execution of individual events, new events are generated by these, thus allowing the perpetuation of the simulation. It is very important the fact that is the same event to generate the new ones, because this produce *causality* relations between the operations. Formally, if an event e during its execution generates an event e', it means that the generated event e' *causally depends* on e. This implies that, for the nature of the causal order between events located in time, if an event e with time T_e generates an event e' with time $T_{e'}$, it follows that $T_{e'} \geq T_e$. Looking to the progression of the execution, this means that an event in the present can not have effects in the past.

An important point is also that, as it happens in the formal DEVS approach, in the systemic one is possible to divide the simulation in two main components: on one side the simulation model, and on the other the simulation kernel on which the model is executed. Moreover there not exist a definition of the standard components of the kernel, rather an indication of a set of base components that are required. Furthermore, as ithappens in event based programming, the whole execution of the simulation relies on a *simulation loop* which advances cyclically the simulation.

Since one of the functionalities of the simulation kernel is to handle event received by the system, every time that the simulation model has to schedule a new event, this is traditionally done using a provided ad-hoc interface. Thus, if during the execution of an event e are generated two new events, namely e' with timestamp $T_{e'}$ and e'' with timestamp $T_{e''}$, and with $T_{e'} = T_{e''}$, there is no order between the two events and then the simulation kernel has to decide which event to deliver before to the respective handler. The problem of events with same timestamp can be faced through the use of a *tie breaking* function, as happens in the DEVS formalism, but this is not trivial since it can affect the behaviour of the simulation model [32].

2.2.1 Basic Components of a Discrete-Event Simulation

In addition to the logic of what happens when events occur, DES includes the following components:

Simulation State The current state of the simulation is shown by its state. A simulation state is a set of variables that capture the main properties of the system studied, explicitly defined in the model code, and altered during

the execution. In fact, as seen before, the state of the simulation is modified during the execution, transiting between states, according to a transition function in the form of $\sigma(s, e) : S \times E \rightarrow S$. The simulation state represents a snapshot of the simulation and it is necessary for a DES model.

Events The evolution of the whole simulation system is carried out by the succession of events. In DES an event is impulsive, then the begin of the event coincides with its end (the duration of the event is zero). All the events that occur inside a simulation have to be defined, and the simulation model have to define a logic associated with each event. This means that for each event received, the simulation model has defined an operation to perform (that is the event handler). Then, the simulation model has to define a closed set of events E, that can be generated/received during the execution and ,for each event $e \in E$, there is an associated event handler defined in the system. Since DES is executed sequentially, all the global variables associated with the simulation state S can be accessed.

At the begin of the simulation, generally one event (called INIT) is injected in the system to allow the begin of the simulation. When this event is scheduled, generally it takes care of the set up of the simulation model, defining the *initial condition*, and then injects the new events to be processed by the system. Normally the INIT event is generated automatically by the simulation core, but since it is closely connected to the individual model, the programmer has to explicitly manage and define its behaviour inside the system.

Clock A simulation model describes the main aspects of a phenomenon in the real world that evolves over time, then the model has to keep track of the passage of time. As seen before, the simulation clock keeps track

of the time inside the simulation, that is completely disjoint by the WCT, thus, due to the discrete nature of the events, its progression during the execution is not homogeneous, but jumps forward from the timestamp T_e of the current executed event e, to the timestamp $T_{e'}$ of the event e' next in order. Generally the simulation clock is kept by a global variable, usually updated by the simulation core, which tracks the temporal evolution. Due to the ordered execution and the causal order between the executed event, this variable never decreases.

Event Queue During its execution, an event e can generate through the simulation model an arbitrary amount of new output events, following the logic of the model. Moreover, since the discrete (instantaneous nature of events, activities that have a duration in the time are modelled as a sequence of event related to the different phases of the single activity (i.e. the begin and the end), which are generated or all together by a previous event, or one by one, during the execution. To keep the generated events, the system needs a data structure. Often this structure is also called *pending event set* because it keeps the events that are pending as a result of the previous executions but have yet to be processed.

Since which each event is associated a timestamp, and since the execution has to schedule the events in time order, the event queue is typically organized as a priority queue, ordered according to event time [33]. Regardless the order in which the events are added to the queue, to determine the next event to process it is sufficient to take the event at the head of the queue, since this is always the one with the lowest timestamp. In case the event e with timestamp T_e is taken after an event e' with timestamp $T_{e'}$ and $T_e < T_{e'}$, it means that the execution has failed. This type of error takes the name

causality error.

The event queue can be implemented in different ways since the more efficient solution depends on the nature of the simulation model. Obviously the aim is to reduce the time needed to insert and get events by the queue. Some of the most diffused solution adopted are linked list, skip list [34], splay tree [35], calendar queue [36], or ladder queue [37].

Simulation Objects This aspect is not necessary for a DES model, then is possible to define simulation models also without simulation objects. However, the use of simulation objects represents a useful extension that gives more semantic capability to the model writer, then the majority of the available simulation kernels support this interesting feature. A simulation object is used to describe a portion of the model, that can be an *agent* (in agent-based simulations), or a *spatial portion* (e.g. a portion of a room, or the cell covered by an antenna). In this way, for the model writer is possible to describe the whole world, concentrating on individual portions, linking them together only at the end, using *interconnection events* (e.g. the transition of a mobile from a cell to another can be modelled like an event sent by the first cell to the second).

Ending Condition Usually simulation models describe phenomena endless and thus could continue their run continuously (e.g chemical reaction, car traffic, network traffic and so on). Moreover, often the models involve stochastic processes to study the evolution of a kind of system, in order to predict how the simulation will evolve. Thus, it is important to well define the moment in which the execution of the simulation has to stop, in order to get meaningful statistics from the experiment. For this reason it is important to define a particular end condition that is verified after each execution of

an events, in order to decide if the simulation has reached its end or goal. The ending condition can be defined in different ways, e.g. it can be a time range of interest for the experiment, or the reaching of a particular value in the simulation state.

Statistics A simulation is carried out to make a prediction and then collect results about its execution. For this reason typically a simulation keeps track of the statistics of the system, which give a feedback on the aspects of interest.

2.2.2 Simulation Kernel's Basic Logic

As seen before, the simulation kernel has the task to carry out the simulation, keeping track of the state of the simulation. First of all the simulation kernel has to set up the environment to start the simulation (done generating the INIT event) and has to manage the generation and the execution of the following events. The structure of a simulation kernel is represented by a main loop, that with each iteration processes the next event in the pending queue. The algorithm of the basic structure, without facilities and optimizations given by the specific implementation, is shown in Algorithm 1.

The skeleton of the Simulation Kernel is composed by two main procedures. The first one is the `INIT` procedure, executed at the begin of the simulation, which:

1. First of all sets the flag *End* to false, that means that the simulation will continue until the ending condition is verified and then the flag is setted to true.

2. Initializes the state of the simulation, that implies the allocation of the memory if this is required by the implementation of the simulation kernel.

Algorithm 1 DES Skeleton

```
procedure INIT
    End ← false
    initialize State, Clock
    schedule INIT
end procedure
procedure SIMULATION-LOOP
    while End == false do
        Clock ← next event's time
        process next event
        Update Statistics
    end while
end procedure
```

3. Initializes the clock used by the simulation, setting its value to zero, otherwise to the initial time of the simulation.

4. Schedules the first event of the simulation, INIT. This operation can be performed either placing the INIT event into the queue, letting its execution to the main loop, or directly executing it. With this operation the initialization procedure ends.

The second procedure, called `SIMULATION-LOOP`, starts right after the end of the initialization, and is composed by two main parts:

1. The first one gets from the event queue the event e_{min} with the smallest timestamp T_{min}, keeping the causal order.

2. The second part updates the value of the clock with the timestamp T_{min} relative to the event e_{min}, in order to keep track of the advancement

of the simulation time. Next, the event is passed to the event handler that processes it according to the logic implemented in the model.

As seen before, an important building block of a simulation model is represented by the collection of statistics, then, at the end of the execution of an event, it is possible to perform the *statistic update*, that e.g. logs information about the performance of the run, or data related to the simulation state. This last point can be intrinsically implemented in the simulation model.

2.3 Parallel Discrete Event Simulation

Discrete event simulation is used to make a prediction of a phenomena in the real world, and often it is used to anticipate the event before it happens. For this reason is useful to reach the end of the simulation soon as possible, then there is the necessity to speed-up it. In this direction, in 1979 Chandy and Misra [38] developed the approach of Parallel Discrete Event Simulation (PDES) [2]. PDES is about the execution of a single DES program on a parallel and/or distributed system, transforming the first one in a new PDES program, that has to give the same results. Since a DES program is already formed by a *kernel* and a *model*, it is possible to do this passage by making just few modification on the model (imposing few restrictions), against substantial alterations in the simulation kernel. This means that, once redesigned the kernel, is possible to port a DES model to PDES, just verifying that it respects the imposed restrictions.

The Simulation Objects, that in DES are used as a non mandatory extension, here take the name of *Logical Processes* (LP) and acquire a great importance for the parallelization, since each LP represents a disjoint portion of the simulation. Thus, in PDES it is possible to say that a simula-

tion is composed by a set of N LPs, each one disjoint from the others and labeled by an unique integer identifier in the range $[0, N-1]$, and called $LP_0, LP_1, ..., LP_{N-1}$. Each LP has a private variable to keep its simulation time. This variable takes the name *Local Virtual Time* (LVT). Its value is different across the LPs and corresponds to the last event executed on each of them.

As hinted before, to develop a model for PDES, we have to impose some restrictions. The most important is about the usage of shared variables: the whole state S of the simulation model has to be divided in sub-portion S_i of the state, assigned as private variables across the LPs (where S_i is the sub-portion of state assigned to LP_i), following the property:

$$S = \bigcup_{i=0}^{N-1} S_i \wedge S_i \cap S_j = \emptyset, \forall i \neq j \tag{2.7}$$

Equation 2.7 tells that, dividing the simulation state in sub-portions of it, the whole state is in the sub-states and, thus, there are no more global variables. Moreover, each LP can only access its variables, thus, the interactions between the different LPs are allowed only by exchanging events.

In some scenario, the elimination of global variables can appear immediate, like in queueing network simulations [39], but it is not so simple in others. An example can be given by the simulation of the movements of people in a room, divided in sub-portions. A variable of the simulation is the number of people in each portion, that changes when someone from a portion to another. To keep track of this variable one can rely on the use of a matrix representing the amount for each portion. However, the update of the corresponding cells of the matrix, can be replaced by an event sent from the starting portion of the room to the one of arrival.

In Figure 2.2 we show a classical distributed architecture for a PDES

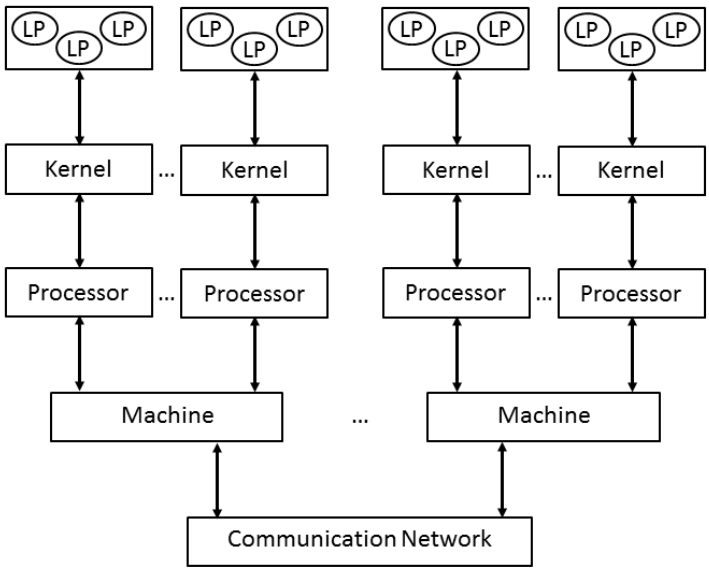

Fig. 2.2: Parallel Discrete Event Simulator Classical Architecture

Simulation kernel. Basically each simulation kernel instance, that runs on a processor as a different user space process, takes care of the execution of a set of LPs. The different simulation kernel instances can be located on different machines and, in this case, are interconnected through a network. While the different instances that runs on the same machine can communicate through different facilities, like shared memory or inter-process communication provided by the operating system, the remote ones can rely on the use of the *distributed memory* paradigm, and on *message passing primitives*, e.g. Message Passing Interface protocol (MPI) [40]. As seen before the communication between the different LPs happens through exchange of events, then at each message corresponds one event. For this reason, the event exchange mechanism often is referred as *message exchange*.

However, during the last years, after the fail of the Moor's Law, multi-core machines have seen a wide spread. To react to this change, and then exploit the available resources given by this new trend in architectures, new

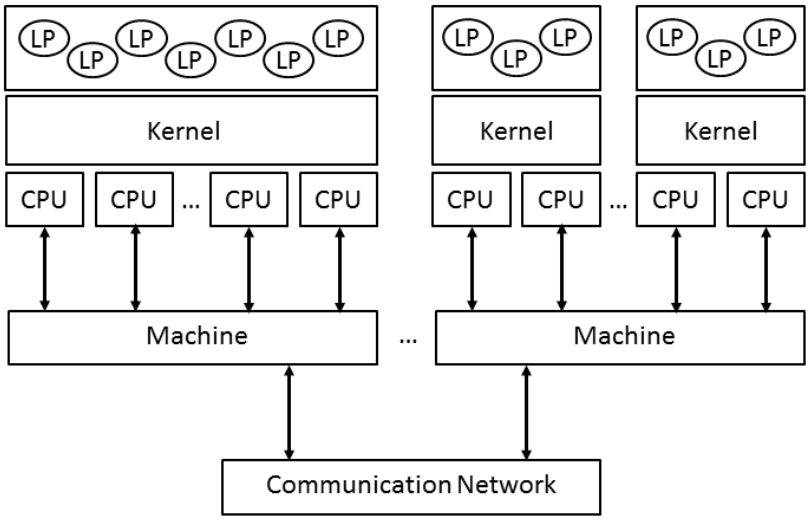

Fig. 2.3: Parallel Discrete Event Simulator Multithread Architecture

paradigms have been studied. A recent research trend, has reshuffled the traditional PDES architecture, moving to a new approach based on a new multi-thread simulation kernel [41, 42, 43]. This will be the reference architecture for this thesis.

This approach, that no more constraints the LP to the single core through a single simulation kernel instances, is shown in Figure 2.3.This new type of architecture has no more a single simulation kernel instance for each processing unit, but each one can run on top of more cores, reducing the number of simulation kernel instances for each machine. To reach this result, the simulation kernel instances are developed using the *worker thread* paradigm. Each thread, similarly to the traditional DES, executes a simulation main loop. The messages, used to carry events, exchanged between the LPs on the same machine, are implemented using the user space, then without the use of external libraries. This new type of approach looks to be more complex and difficult to implement, but has been proven to introduce an improvement in

scalability and performance if compared to the classical PDES implementation [44].

In order to fully exploit the locality of data, it can be very useful to associate symmetrically the management of the LP to a single worker thread. This concept is called *binding* [45, 46]. To do this, are defined temporal windows inside which a thread is statically associated to a local available LP. The duration of this window can last for the whole execution, with a *static binding* implementation. Otherwise, the duration of this windows can be variable during the execution, recomputing them periodically, e.g. with the aim to balnce the workload across the worker threads. This technique take the name of *load sharing* [44, 45, 46], different from the *load balancing*, used in the traditional PDES to migrate an LP from an instance of the kernel to another one.

As seen before, each worker thread executes a main simulation loop and maintains its local state variables, including the local time reached by the simulation on each single LP. This allows a parallel execution of each worker thread regardless the other ones. Let's see what can actually happen in this scenario. Let's consider the situation at a given wall clock time of a single simulation kernel instance. Let's consider a logical process LP_i, bound to the worker thread k_0 with simulation time $LVT_i = 3$, and a logical process LP_j, bound to the worker thread k_1 with simulation time $LVT_j = 9$. Relying on the most common implementations of simulation kernels that make use of multiple queues following Algorithm 1, both of them will get respectively the minimum events e^i_{min} and e^j_{min} as next events. Assuming that the events e^i_{min} and e^j_{min} have respectively timestamps $T_{e^i_{min}} = 6$ and $T_{e^j_{min}} = 12$, the processing of these will bring the local virtual time of the LPs respectively to $LVT_i = 6$ and $LVT_j = 12$. Now, if the execution of the event e^i_{min} generates

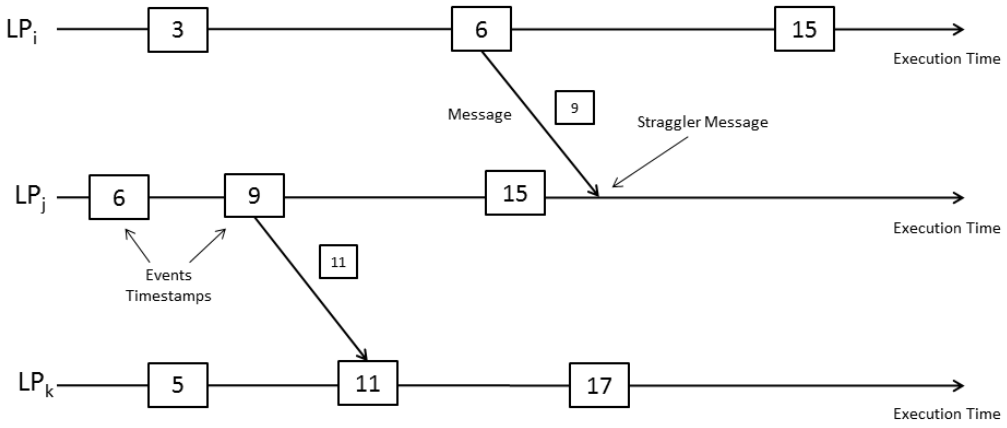

Fig. 2.4: Example of Event Causality Violation

a new event e_{new} with timestamp $T_{e_{new}} = 9$ with destination LP_j, what happens is that LP_j, that has clock equal to $LVT_j = 12$, receives and has to execute an event with a timestamp lower than the current time. The event e_{new} takes the name of *straggler message* and represents a causal violation introduced by the parallel nature of PDES. This scenario is represented in Figure 2.4.

2.3.1 Synchronization Problem

As shown in Figure 2.4, the parallel LPs in PDES can produce causality errors due to their asynchronus execution. This problem takes the name of *synchronization problem*.

To face this problem, with the aim to guarantee the correctness of the execution independently of the behaviour of the different LPs, are available different type of approaches [47, 34], divided in three main categories: *conservative* [48, 38, 49], *optimistic* [3] or *hybrid* [50, 51] approaches. The conservative approach faces the problem simply avoiding a-priori the possibility

of a causal violation, executing an event only when it is considered *safe*, then if it is definitely the next one to execute. The optimistic approach instead allows the execution to go ahead, verifying a-posteriori if there are some violations and, in this case corrects the error introduced. The third approach makes use of both the conservative and optimistic approaches, using at each time the best suited approach for the single situation.

2.3.1.1 Conservative Approach

The conservative approach was the first synchronization approach developed to guarantee the causal order in PDES systems. This is a pessimistic approach because prefers slowdown the execution rather than trying to execute some unsafe events. However this is also the simplest to implements. In facts, the goal of this approach is to decide if the event taken from the pending queue of an LP is safe with respect the others LPs. This means that the execution progresses by processing the event e_{min} with the smallest timestamp [52].

A first approach [48, 38] compelled the model writer to specify explicitly where each LP communicated with the others, defining statically the communication channels. This means that each LP has more than a channel, seen as a FIFO, and with each one is associated a timestamp. The timestamp associated to a queue is, the timestamp of the first message of the queue (the smallest one), or the timestamp of the last event executed (if the queue is empty). Each LP gets the next event to be processed by the queue with the smallest timestamp, or, if the queue that has the smallest timestamp is empty, waits until it receives an event in the selected queue.In this last situation the LP waits for a new event in the empty queue because, if it takes an event from another queue with larger timestamp, it may eventually receive a

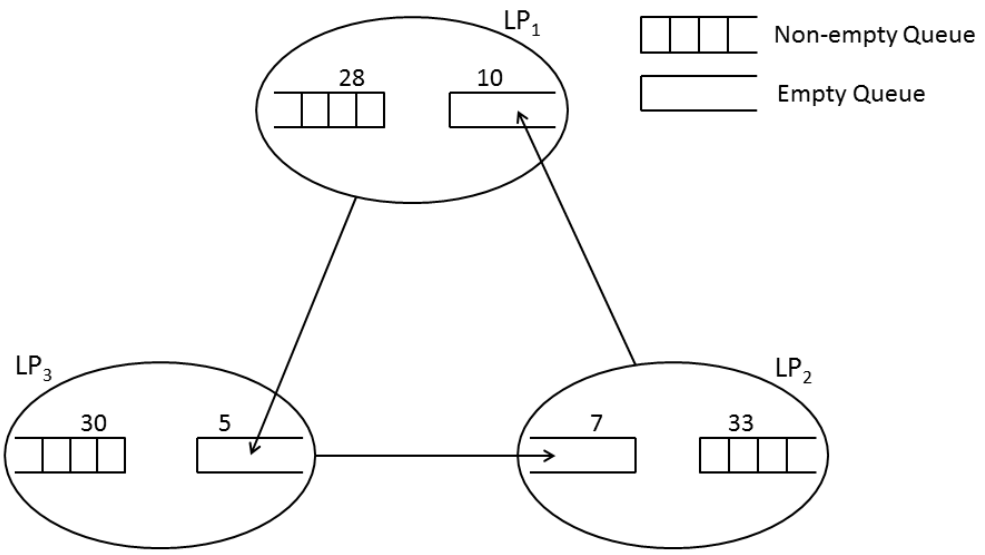

Fig. 2.5: Deadlock in Conservative Synchronization

smaller event in the queue with the minimum timestamp, not violating the FIFO order of the queue itself, but violating the causal order of the whole execution. Thus, the waiting approach is a necessary and sufficient condition to guarantee the correctness about the causal scheduling of the events.

However, this approach can easily lead to deadlocks. Let's assume a simulation model that follows the scheme in Figure 2.5. LP_1 has a channel of communication with LP_3 that, in turn, has a channel of communication with LP_2 that has a channel with the first, and each one of the three LPs has another channel of communication. As in the schema, all the queues associated to the three channels are empty, then there is a deadlock situation. This happens because, also having an additional queue that continues to store incoming messages from others LPs, the algorithm continues to check the empty queues because it has the lowest timestamp.

This deadlock situation can be avoided by using *null messages*. These

are messages that are sent to the others thread but that do not carry events to deliver. When a null message with timestamp T_{null} is received, this means that the sender will send no next message with timestamp $T < T_{null}$. In this way the system is able to determine that LP_i will not receive any event e_k with timestamp $LVT_i \leq T_k \leq T_{null}$ and then has the possibility to execute any event e_j from a different queue with a timestamp $LVT_i \leq T_j \leq T_{null}$ since it is considered safe.

Another important aspect that can have non-negligible effects on the performance of the system, is the amount of minimum simulation time that elapses between an event and the one generated from it [53]. This value takes the name of *lookahead* and can be used to determine which events can be concurrently executed. Considering that LP_i has reached the minimum simulation time LVT_i in the whole system (i.e. the minimum clock between all the LPs of the simulation), it can execute all the events e with a timstamp T_e such that $LVT_i < T_e < LVT_i + L$. At the same time, each other LP_j that has reached a clock LVT_j such that $LVT_i < LVT_j < LVT_i + L$ can execute any event e' with timestamp $T_{e'}$ such that $LVT_j < T_{e'} < LVT_i + L$. This obviously implies that, larger values of the lookahead entail an higher probability that an event is considered safe and then can be executed without being the smallest timestamp one. However, the lookahead can not be randomly setted, then can not be big as we want, but has to be exactly calculated and included by the model writer. In fact a value too big could affect the correctness of the simulation.

On the other hand, a value too small, or even zero, for the lookahead, introduces a significant slowdown of the execution, removing the gain given by the usage of one or more multi-/many-core machines. In [54] can be found a proposal studied with the goal to increase the lookahead value, pre-

computing a section of the pending events chain, and analysing at the end of this computation if the resulting state is still consistent.

The conservative synchronization approach provides three main advantages:

- *Aggressiveless*: the simulation is executed in manner that in each time of the simulation the state is consistent. This is done ensuring that the execution of each single event does not cause error conditions;
- *Riskless*: the results entered in the system are ever related to a consistent part of the simulation execution, then all the data sent to other components of the model are correct;
- *Minimal Synchronization* among LPs: the simulation, differently by the optimistic synchronization approach, strictly increase, because the execution is never rewound. In this way, to detect the global time reached by the simulation it is ever a simple task to execute and do not incurs in high cost operations.

However, the conservative synchronization approach in most cases is not able to fully exploit the parallelism provided by a multy-/many-core architecture or distributed system. In fact, taken two events e_1 and e_2 in the system, the conservative approach will verge to a sequential execution of these also if there is not any correlation between the two events.

For this reason, during the last years, the research in the parallel paradigm of the simulation has focused on the developments of optimistic approaches. The idea of this kind of approach is that, instead of waiting for the assessment of event safety, it makes a bet choosing the likely correct path to follow. For this reason this technique is referred also as *speculative approach*. My thesis

uses this type of approach to exploit the facilities given by a particular type of hardware. In the next chapter we detailedly decribe the speculative approach (i.e. optimistic synchronization) used to develop *Speculative Parallel Discrete Event Simulation* platforms.

Chapter 3

Speculative PDES

As seen at the end of the previous chapter, the adoption of a conservative synchronization policy can lead to limited exploitation of the resources given by the underlying architecture. For this reason during the last years the research in parallel discrete event simulators has focused on the development of optimistic protocol implementations. The optimistic synchronization is based on the *speculative processing* approach. This technique is used in many context, especially where there are resources unused due to some wait periods. One of the most important example is given by the pipelining used in the processing units [55] where the speculation is used to fully exploit all the components of a CPU without waiting the result of the previous instructions. Other important examples are given by file prefetching [56], and by the concurrency control in transactional systems [57].

This technique is based on the idea that, if there are resources that are available, rather than leaving them unused, these should be used to execute tasks that are *likely* correct or/and useful. Basically there is a bet done before on the fact that this work can be really useful, discovering only later if it was actually needed, then if the gamble paid off. In case that later the

work done is discovered to be not useful or incorrect, the results obtained are discarded and the execution continues its correct flow. The main advantage of this approach is due to the fact that the resources used to do the extra work related to the speculation would be not used otherwise. Differently, waiting to know the result of the previous executions to decide what to do might introduce a delay in the execution of the whole task. If the speculation done is correct this delay is eliminated. On the other hand, if the speculation is wrong, the system needs to discard the results of the part of work related to the speculation.

In PDES environment, this technique was introduced for the first time, under the name of *optimistic synchronization*, in the work published in [58] , where was introduced the mechanism called *Time Warp*. The optimistic synchronization, differently from the conservative synchronization, on each simulation kernel instance and for each worker thread, selects locally the smallest event in the queue as the next event to be processed independently from its safety. In this way, it is obvious that the guarantees given by the conservative approach about the correctness are lost, and then the simulation can fall into an inconsistent state. In fact, as shown in the previous example in Figure 2.4, now the simulation can follow a trajectory affected by a causal violation, reaching in this way an inconsistent simulation state.

3.1 Recoverability

In the speculative approach then, the execution is carried out choosing at each time an event with the smallest timestamp from the local queue, without considering possible causal violations. This requires a mechanism able to detect a-posteriori the reception of a out of order message (i.e. which has

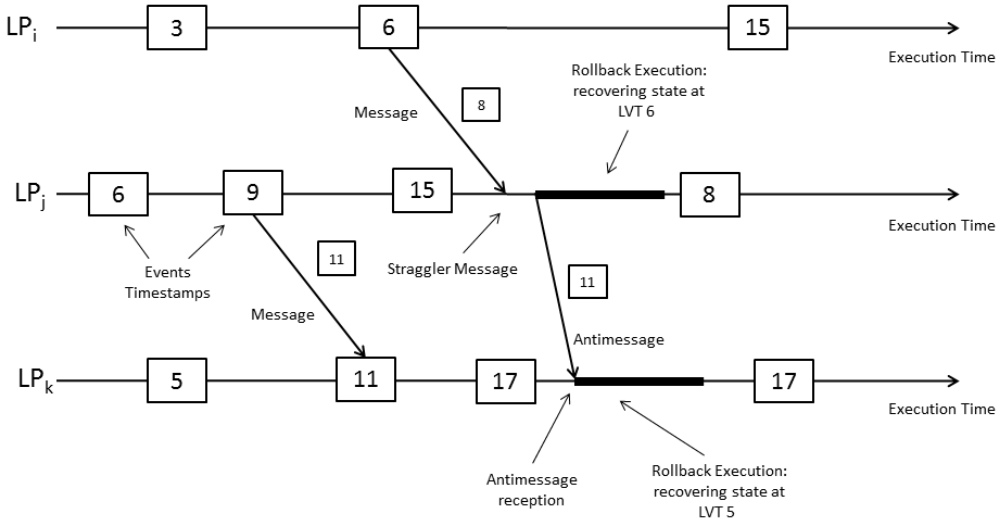

Fig. 3.1: Rollback Operation

a timestamp smaller than the local clock) with the relative occurrence of a causal violation, to temporarily stop the execution on the simulation model, to go back until a previous consistent simulation state is found the simulation time, and then to resume the the normal execution flow starting from the last received message that has triggered this mechanism. This operation takes the name of *rollback operation*, and is shown in Figure 3.1.

In the example it is possible to see that executing a rollback operation, gives rise to another problem. In particular, when LP_j receives straggler message e_{str} associated with timestamp $T_{str} = 8$, its clock is $LVT_j = 15$. Therefore, it rollbacks the execution to the simulation time $LVT_j = 6$, i.e. it then discards the changes of the simulation state due to the events with timestamp $T = 9$ and $T = 15$. This operation is sufficient to restore a consistent state for LP_j, since the portion S_j, that is a part of the global state, is correctly rolled back to time $T = 6$ that is consistent, also with respect the new message received. However, during the execution of the

event flagged with timestamp $T = 9$, LP_j has sent a message associated with an new event with timestamp $T = 11$ to LP_k. This message, that was considered correct until the reception of the event e_{str}, is now revealed as wrong result of an incorrect simulation trajectory, given that it was sent by LP_j without considering the flow of LP_i. Since LP_j has rollbacked to the previous time $LVT_j = 6$, it is possible that the execution of the new message e_{str} changes the trajectory of the simulation and then, may be that the message with timestemp $T = 11$ should have not be generated and sent, or it should carry a different information. Then, the situation is:

- LP_i at $LVT_i = 6$ has sent the straggler message e_{str} to LP_j;
- LP_j rollbaks to simulation time LVT_j such that $T_{str} > LCT_j > T_{e_{max}}$, where $T_{e_{max}}$ is the event already processed with the largest timestamp before e_{str};
- this implies to discard the event processed by LP_j which has caused the generation of a new event sent to LP_k.

Then the rollback operation that has involved LP_j, produces in turn an inconsistent simulation state also for LP_k. Then it is needed that LP_j notifies to LP_k the changes due to the rollback. This means that the system has to support a mechanism to notify to LP_k that an event received by LP_j is no more in place, than that its execution might carry the system to an inconsistent path in the trajectory of the simulation. This operation is done using *antimessages*. An antimessage $\overline{e}$ is the negative version of an already sent message e, sent by some LP to another one.

Then, in the moment that LPj has to rollback, it checks if it has sent some positive messages during the interval $[LVT'_j, LVT_j]$, where LVT_j is the simulation time reached when the struggler event e_{str} is received, and

LVT'_j is the simulation time to which the simulation is restored. For each message produced in this interval, an associated negative message is sent to the respective LP.

When LP_k receives an antimessage, there are two possible scenarios:

1. the message e associated with the antimessage $\overline{e}$ has a timestamp T_e such that $T_e > LVT_k$. This means that the message e has still not processed (otherwise the simulation time LVT_k would have been equal or larger than T_e). In this case the only effect of the processing of the antimessage is to annihilate the positive message, then e is simply removed by the pending queue of LP_k;

2. the message e associated with the antimessage $\overline{e}$ has a timestamp T_e such that $T_e \leq LVT_k$. This means that the message e has already been processed by LP_k, and then LP_k has executed an inconsistent path of the simulation trajectory, reaching an inconsistent simulation state.

In this last case we have that LP_k has to rollback to a consistent simulation state (as happens in the example given by Figure 3.1). The fact that the rollback of some LP might cause the rollback of other LPs, is called *cascading rollback* and can affect multiple LPs.

It is important to say that the rollback operation can be a feature implemented entirely in the simulation kernel. In fact, if the simulation kernel knows the location in memory where the simulation states S_i of the various LPs are stored, it can manage the rollbacks operation without the need to modify the simulation model. However, the programmer has to explicitly indicate where is located the simulation state in memory.

In literature there exist two main approaches to support rollback operations. The first one relies on the concept of *save state & restore*, and it

is the original rollback approach proposed in [58]. The second one relies on the concept of *reverse computation* of the simulation events executed, and assumes that, starting from an event, it is possible to generate a *negative event*, able to completely execute the same operation of the original event, but in reverse order and with inverse effects. Now it is important to distinguish the concepts of *antimessage*, used by an LP to notify to another one that an event sent has to be completely annihilated (both if processed or not), from a *negative event*, that may be triggered on the reception of an antimessage. and that undoes the effects of the execution of an event on the simulation state.

Before discussing the two aspects it is important to introduce the concept of *progress condition* in the speculative approach and how to prove it.

Global Virtual Time (GVT) If on one hand it is easy to prove that the rollback operation is correct [59], on the other hand it is not so easy to prove the progress condition of the whole system, due the presence of the rollback operations themselves. Moreover, since the events to be executed might be part of a trajectory that will be discovered later to be inconsistent, it becomes difficult to decide when and how to check when the ending condition has been reached. Still, can be very difficult to handle error conditions and I/O operations.

The original proposal in [58] proposes to keep track of the progress in the whole system execution, using a global mechanism to control the evolution of the whole simulation time, based on the concept of *Global Virtual Time* (GVT). The GVT is a property of the system in an instantaneous snapshot of the global system, taken at time t of the wall clock time and so defined:

Definition 3.1.1. (Global Virtual Time). GVT(t) is the minimum times-

tamp of any unprocessed message or anti-message flowing in the system at WCT t.

Definition 3.1.1 tells that the value associated with the GVT can be computed at any WCT t just controlling the timestamp of all the event not jet processed, included the ones still carried by messages, but it does not tell where these events are and how to do it. And while could be easy to get the minimum timestamp from the messages stored in the pending queues of each simulation kernel, it is not so much easy to inspect the transiting messages, then the ones sent by LP_j to LP_i but not yet received (e.g. due to transmission latency between two remote simulation kernel instances). This last kind of message takes the name of *in-transit message*, and has to be considered as well as the ones stored in the queues.

This is not the only definition of GVT, other ones are referred to different implementations, depending on the different simulation scenario. However, in all the different scenarios, computing the GVT value is an essential aspect of a distributed PDES, since it is required to define the *commitment horizon* of the simulation. Then, taken the GVT(t) at the instant t of WCT, corresponding to the timestamp of the smallest event in the system not yet processed, it is clear that no execution can generate a new event e' such that $T_{e'} < GVT(t)$.

Then this property tells that at an instant t of WCT, no rollback operation can bring any LP_i to a simulation time $LVT_i < GVT(t)$. This means that all the the executions of events with timestamps T such that $T < GVT(t)$ can be considered *committed*, and then can be used to verify the ending condition of the simulation.

3.1.1 Rollback Supports

As mentioned before, to support the rollback operation there exists two main approaches: *save state & restore* and *reverse computation*

State Save & Restore

This approach was proposed for the first time in [58] with the presentation of the Time Warp protocol. This technique relies on the idea that can be taken a *snapshot* of the simulation state of LP_i (i.e. a copy in a different buffer of the memory regions that contain the private state variables associated with the sub-state S_i), associating it with the timestamp of the last event executed. This operation can be done in a transparent way by the simulation kernel, or can be performed with the support of the programmer that has to specify the portion of memory region where the simulation state S_i is stored, depending on the programming model used by the simulation kernel.

In the moment that a rollback operation is triggered, it is defined the simulation time $T_{rollback}$, and then the restoration of the previous correct state can be simply done by retrieving the snapshot of the state S_i associated with the timestamp $T_s = T_{rollback}$, and replacing the current values of the selected sub-state. It is important to note that, when a straggler message is received, to reach the state at time $T_{rollback}$, it is possible to select any simulation snapshot such that $T \leq T_{rollback}$, since it represents a correct state with respect to the new received straggler message. However, taking the most recent snapshot with timestamp smaller than or equal to $T_{rollback}$ will waste minimal work.

Take a complete snapshot of the simulation state of an LP is a costly operation, since it requires both an high amount of memory and time to be performed. From the side of memory footprint, to reduce the amount of state

saved, it is possible to rely on the previous description of the GVT as given in Definition 3.1.1. Since the states stored are used only to support rollback operations, starting from the Definition 3.1.1 we know that at WCT t it is possible to rollback only up to a previous simulation state such that $T_{rollback} \geq GVT(t)$. This means that at each WCT t it is possible to discard all the saves snapshots of simulation state s such that $T_s < GVT(t)$, in a way to free the used memory buffers so as to reuse them in the future. This operation is referred to as *fossil collection* [60] and can be executed periodically to recover memory. Obviously the fossil collection operation can be performed after a new GVT value is computed. Then, the frequency with which the $GVT(t)$ value is computed, and consequently the fossil collection operation is performed, can be manually introduced by the model programmer, or can be computed depending on the hardware architecture used to perform the simulation [61]. However, since the computation of the GVT requires non negligible effort by the whole simulation system, incurring in decrease of the simulation throughput, the frequency of fossil collection is performed has to be balanced with respect to the throughput of the system and the memory usage.

On the other hand, to reduce the cost in taking a new snapshot, it is possible to change the grain with which this operation is performed. At the same time, reducing the frequency of the state saving operation, leads to reducing memory usaged. Another measure that can be taken to face both the problem of the execution cost and memory usage, is to change the amount of information stored during a state saving operation. In fact, often the most part of the simulation state is not touched and then it is possible to save only a portion of the state, then a snapshot can be $full$ or $incremental$.

This is the solution that I followed during my thesis, focusing on small events, trying to reduce as much as possible the amount of data stored, exploiting facilities offered by transactional hardware.

Reverse Computation

While the *save state & store* can be considered a static and discrete approach, since it relies in a periodical copy of the simulation state, the *reverse computation* [14] gives a completely different solution to support rollback operations. This approach relies on the concept of *reverse event* to restore a previous simulation state. The idea is that, with the help of compiler techniques, it is possible to automatically create, for each (positive) event, a reverse event that executes in reverse order operations able to undo the effect of the corresponding forward operation. In this way, executing a reverse event we can undo the effect of the correspondent one. Thus, to rollback the simulation state LP_i that has simulation time LVT_i to a simulation time $T_{rollback}$, the rollback operation has to execute in backward order all the reverse events from LVT_i to $T_{rollback}$.

With a simple approach, a reverse event can be seen as a copy of the regular event, which executes the inverse of the same operations but in reverse order. An example of a simple reverse event is given in [14] where the authors model a transition on an ATM multiplexor model:

```
1 if (qlen > 0) {
2     qlen--;
3     sent++;
4 }
```

The generated reverse event should be:

```
1 if (qlen "was" > 0) {
2     sent--;
3     qlen++;
4 }
```

It is possible to note that, while generating a reverse instruction of an arithmetic operation can be simple to do, this is not the case for branching conditions, since it implies to check an old variable that is no more available when the reverse event is processed.

Then, to face this problem, *bit variables* are used to modify regular events. These are added transparently to the events to notify if a branch is taken or not during the regular execution of the event. Then the previous code snipped becomes:

```
1 if (qlen > 0) {
2     b=1;
3     qlen--;
4     sent++;
5 }
```

Now, the bit variable is set to 1 only if the branch condition is verified, then the reverse event can rely on this value to verify this and then execute or not the reverse body of the branch:

```
1 if (b == 1) {
2     sent--;
3     qlen++;
4 }
```

This kind of approach, introducing a new variable for each branches, increases the size of the simulation state of $\log(\#branches)$, but, on the other

hand, the introduction of this bit, simplifies the management of condition branches in reverse events. Moreover, the bit variables can be used to manage `switch/case` constructors.

Another problem is represented by the fact that unfortunately not all the operations are reversible. For example there are operations that destroy all the information on the original value, referenced as *disruptive operations*, e.g `=` or `%=`, that have to be managed using the state save technique. Moreover, since these operations are characterized by a fine granularity, and since the original simulation model's executable is instrumented to modify it to support the generation and the execution of reversible event, is possible to do this by relying on the approach in [62] cited above, where the changes are tracked generating an incremental fine-grain snapshot for each memory update.

The loops can be simply managed by executing n time the reverse code inside the loop, taking the information about the number n of iteration in the original execution, that is very important in case of exit loop condition, like with the `while` statement.

However, the management of jump instructions (e.g. `break`, `goto` or `continue` instructions) or call to functions require the use of some bit variables to remember the flow of the original execution. Then the execution of the reverse events can rely on a set of auto-generated `switch/case` to be able to reproduce the reverse execution flow. However, the insertion of this new portion of code can give rise to a non negligible increase of the state size, depending on the execution flow of the simulation model and on the complexity of the event's code.

An additional important part that deserves particular attention in reverse computation is about pseudo-random number generator. To guarantee a

deterministic execution of the flow, there is the necessity that multiple calls to the random number generator, relative to the same logical invocation, return the same value. This means that, if the same portion of simulation is executed multiple time, starting from the same initial situation (i.e. if is considered an execution rollbacked multiple times without introducing changes in the flow), the results have to be ever the same. This can be done applying the mechanism used to generate reverse events also to the code of the random number generator, assuming that there is no loss of information due to lossy floating point operations.

Anyway, reverse computation can give an important reduction in the time needed by rollback operations, especially if the state to be rolled back is near to the actual logical time of the LP, and it is done adding a limited overhead. On the other hand, if the simulation code executes a large number of disruptive operation, it can fall back in the case of fine-grain state saving, but still maintaining a lower overall performance.

3.1.2 Additional Components of Speculative PDES

In the previous Section 2.2.1 the most important components of DES have described, but now it is obvious that to support a parallel version of DES these are not enough, especially considering the use of optimistic synchronization approach. In fact, while the conservative approach can relay on one or more event queues, the Time-Warp based simulation kernel has to store, in addition to all pending events, also all processed events (since there is the possibility that the kernel has to execute again a portion of the processed trajectory), and simulation states. The essential building blocks of a reference architecture to support the speculative synchronization protocol are shown in Figure 3.2.

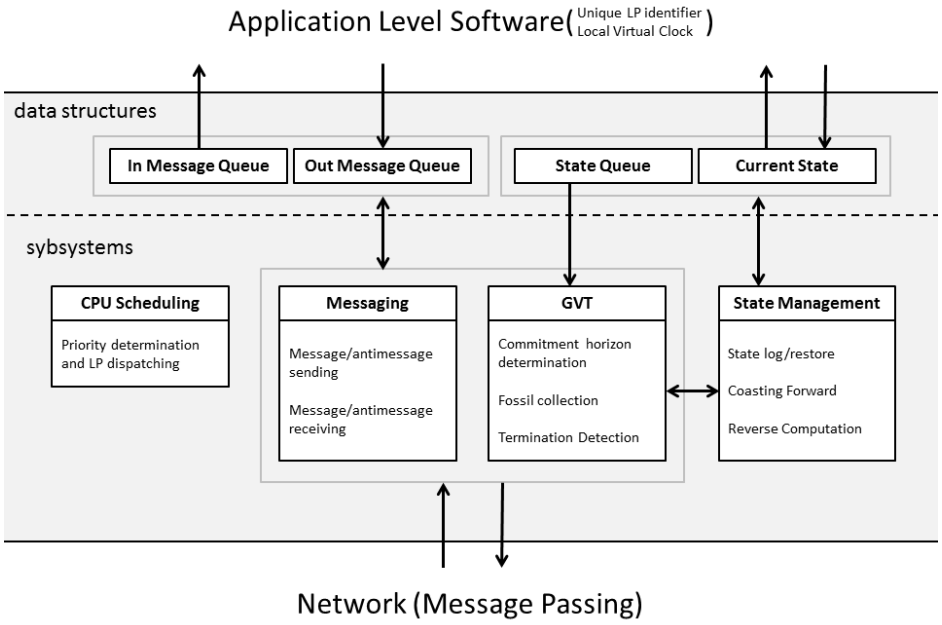

Fig. 3.2: Reference Architecture for Optimistic Simulation Systems

Input and Output Queues As sad, in the Time-Warp based approach, it is necessary to keep track also of the processed events, then, in addition to the (input) pending event queue, we find an output queue, in order to store old messages with the relative destinations (that is mandatory since during the rollback of an single LP, it has to send antimessages to annihilate the effect produced by the inconsistent trajectory). Many implementations, to keep the information about the destination, rely on multiple output queues, one for each other LP, in a way that, when scanning the output queue to generate antimessages, the LP has to check only the relative timestamp since the destination is identified by the queue itself.

Messaging Subsystem While in sequential DES implementations the single simulation kernel is obviously stored in a single machine, PDES gives the possibility to distribute the simulation kernel across more machines. This

means that LPs are hosted not only locally, but also on remote simulation kernel instances, introducing the need for a routing subsystem that takes care of messages. The use of a sophisticate message routing system gives to the model the possibility for LPs to communicate without taking care of the location of the relative kernel instance, then using simply uniform API for messages' scheduling. It is the message subsystem, that is a part of the simulation kernel, that will determine where to send the message. Moreover, the management of the output event queue is often assigned to the message subsystem, such that the antimessage-sending procedure can be decoupled from the execution of rollback operations.

State Queue & State Management Subsystem The original technique to implement the rollback operation was the *statesave&restore* one. This technique relies on a buffer to save states during the simulation. To fulfil this task a *state queue* data structure is exploited.

The *state queue* data structure is handled by the *state management subsystem*, which takes care of:

1. maintaining an ordered list of stored states organized following the timestamp of the snapshot, and adding the new snapshot taken by the system;

2. performing the rollback operations determining the state to restore from the queue, or also performing reverse event until the rollback time;

3. performing *cost forward* operation (snapshots are periodically taken then, once one is loaded, the system has to re-execute some events to reach the exact point in the local simulation time);

4. performing fossil collection operations (i.e. periodically has to compute the GVT(t) and delete events and state logs belonging to the already committed portion of simulation, in order to recovery memory).

GVT Subsystem The *Global Virtual Time subsystem* periodically queries the message queues and the messaging subsystem (to not lose transiting messages) in order to execute a global reduction across the whole simulation system to compute the new GVT value and then correctly define the commitment horizon. Moreover this subsystem has also the task to check whether the termination condition is verified and then the end of the execution is reached. This task is performed by this subsystem since the check has to be done on the commitment horizon. In addition, as said in the previous paragraph, the fossil-collection operation has to be performed upon GVT calculation, then also this task is assigned to the GVT subsystem.

Event Scheduler A central problem is related to the CPU-scheduling approaches used to select which LP, in the set of the ones kept by a given simulation kernel instance, has to be scheduled to process its event. As sad before, the common choice is to follow the Lowest-Timestamp-First (LTF) algorithm [63]. This approach selects the LP that has to execute the event with the smallest timestamp between the ones hosted by the same kernel. With this approach the possibility to generate an event that violates the causal order between the LPs hosted on the same kernel is avoided. This happens because the LPs are dispatched following what would happen in a sequential simulation engine. This means that rollbacks can be generated only by events received from LPs hosted by others kernel instances.

Random-Number Generator The optimistic synchronization, in both its approaches, requires that the values produced by a pseudo random number generator are deterministically produced. For this reason the simulation kernel has to provide a random number generator able to support rollback operations and , if rollbacked itself, to produce the same sequence of "random" values. This can be done either storing the seed associated with a snapshot (then, each time a snapshot is taken, also the value reached by the random number generator has to be stored), or implementing a generator able to undo the internal state of the random generator itself.

3.2 Software Solutions

The majority of the solutions proposed to implement PDES platforms and face the recoverability problem are entirely based on software facilities. In this section we provide an overview of the most relevant solutions. The attention will be focused on *state saving & restore* techniques, since this is the branch of recoverability followed by my thesis.

3.2.1 State Saving & Restore

As mentioned, *state saving & restore* is a technique used in speculative PDES to address the problem of the recoverability of a previous consistent state. As seen in Section 3.1.1, there are different ways to take snapshot of the state (and consequently restore it), with different performance in terms of time and space.

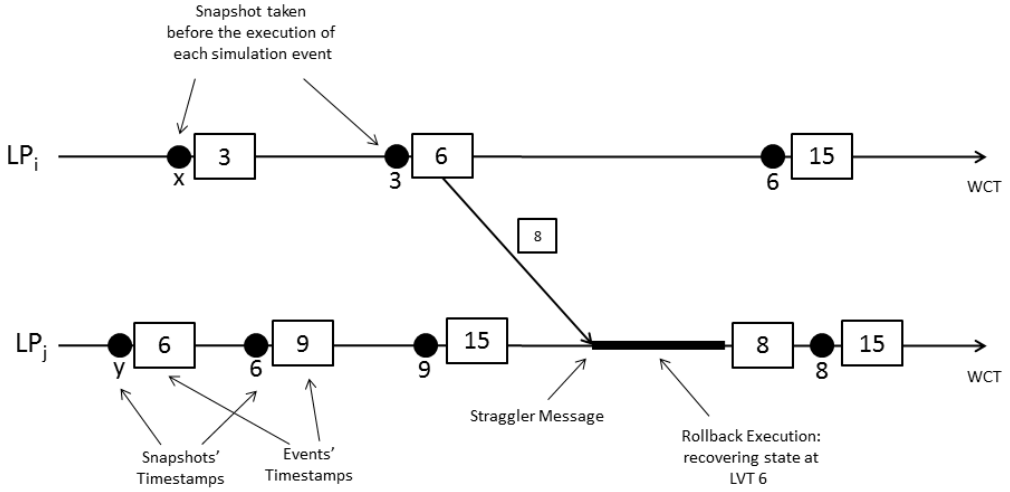

Fig. 3.3: Copy State Saving Approach

3.2.1.1 Copy State Saving (CSS)

This is the first solution proposed to face this problem in [3], and this is also the simplest idea to implement. As show in Figure 3.3, CSS takes a complete snapshot of the state S_i just before the execution of a new event.

This operation is done for each event processed then, to restore a consistent state at simulation time $T_{rollback}$, CSS has simply to take the snapshot S_i corresponding to a T_s such that $T_{rollback} > T_s$ and there is no executed event e such that $T_{rollback} > T_e > T_s$.

However, takeing a snapshot before each event, is not a tenable approach in terms of computing power and memory usage. Moreover, this lead to the need for increasing the frequency of the fossil collection operation (together the GVT computing), reducing even more the whole performance.

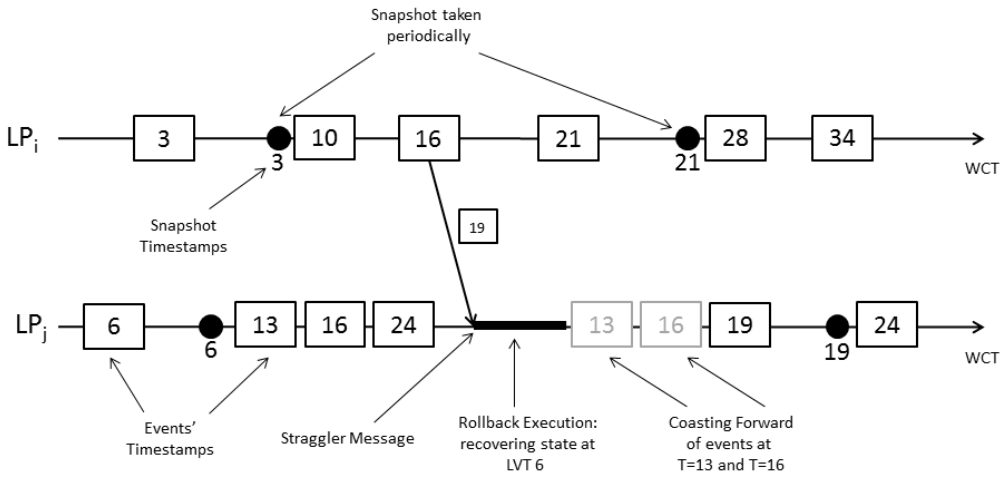

Fig. 3.4: Sparse State Saving Approach

3.2.1.2 Sparse State Saving (SSS)

To face the problem associated with the previous technique, various approaches have been proposed, all gathered together under the class of Sparse State Saving (SSS). The idea of SSS is to decrease the frequency with which the snapshots are taken, rather than performing this operation at each event. Namely the snapshot are *sparsely* [64] taken, as shown in Figure 3.4.

The period between two consecutive snapshots can be fixed, and this is the case of *Periodic State Saving* (PSS), or variable, as happens in *Adaptive State Saving* (ASS).

Now we have no more a snapshot for each event, so, when a rollback operation is performed, there are two possible cases:

1. There is a snapshot s associated with timestamp T_s that corresponds exactly to $LVT_{rollback}$, and then it is sufficient to restore it, as it happens in CSS.

2. There is no snapshot that corresponds exactly to the timestamp of the

recovery point.

In the second case, it is restored a state s associated with timestamp T_s such that $T_s < LVT_{rollback}$, and after intermediate events are executed until $LVT_{rollback}$ is reached. This last operation is referred to as *costing forward*. Since the rollback operation has to be performed only until $LVT_{rollback}$, the system has to send antimessages only for the events e such that $T_e > LVT_{rollback}$. This means also that the system has not to send again message produced during the coasting forward, since these events have already been processed and the produced messages have been already sent. The execution of events without the production of output messages is called *silent execution*.

Moreover, since during coasting forward the system re-executes events belonging to the original (correct) trajectory, it is important to have a deterministic behaviour in their execution, then all the random operations or probabilistic distributions have to follow the same logic, so as to produce the same result. A different result would be wrong either for the logic of the simulation and for message based interactions. Then, in the moment that LP_i reaches the straggler message, it must have rebuilt exactly the same simulation state before performing $e_{straggler}$. This behaviour is called *piece-wise determinism* (PWD).

If on one hand SSS reduces the memory consumption (and the relative operations), on the other hand it increases the overhead due to the coasting forward operation. These two aspects are strictly linked to the frequency with which checkpoints are taken, i.e. the checkpointing period χ.

In the ASS approach this problem is faced using a run-time fine-tuning of the variable χ to follow the changes in the dynamics of the system, with the goal to maximize the whole performance.

3.2.1.3 incremental State Saving (ISS)

However, to face the problem of the excessive memory usage in state saving, regulating the checkpointing interval is not the only available approach. SSS aims to reduce the time needed to perform a rollback operation, without considering that the store/load operation itself can be heavy due to the amount of data that compose a state S_i. Moreover, often between two checkpoint the really modified informations is only a small portion of all the overall states, then a lot of time is spent to copy redundant data.

Incremental state saving aims to reduce the size of the single snapshot by storing only the modified portion of the state, reducing also the execution time to perform the save and restore operations.

Incremental state saving was introduced for the first time in [65]. The approach proposed in this first dissertation relies on events extended with some additional informations:

- The value of state variable modified after the event is executed;
- The value of state variable modified before the event is executed;
- The simulation time T_{gen}, that is the timestamp of the event that generated it;
- The simulation time T_{exe}, that is when the event must be performed, with $T_{exe} > T_{gen}$.

Then a new queue is used to store the modification introduced by events executions. The elements of this queue are linked to the corresponding events that caused the changes in the state. To this new approach to store the old states of the simulation, we associate a new way of restoring the state. When LP_i receives a straggler message $e_{straggler}$ with timestamp $T_{straggler}$,

the system scans all the processed events such that $LVT_i \geq T_{exe} \geq T_{straggler}$, and restores the correspondent state variable when modified (taking care to replace each variable once).

This approach is not transparent, since the model writer must have knowledge about rollback and state saving concepts, and has to access structures of the kernel to save state variables before modifying them. This is necessary since the kernel does not know where the state is stored, and which variables are being updated.

For this reason, many solutions have been studied to develop approaches able to provide transparency to the model writer.

The work in [62], studied for x86 architectures, provides transparency of the state saving operations using software instrumentation. This approach starts by the assembly code of the simulation model and, parsing it, each time an instruction that performs a memory update is found, it adds a `call` to a module that makes a copy of the old value stored in. When a straggler event arrives triggering a rollback operation, the chain of updates is backward scanned, realigning the memory to the previous consistent state. This technique is completely transparent for the programmer that has not to modify the original model, since this mechanism is automatically supported by the simulation kernel.

Also the approach discussed in [66] is based on the instrumentation of the assembly code of the application. However, this time the old value of the updated memory portion is not directly saved. This approach relies on a *dirty bitmap*, that is a structure where, each time that a memory region is updated, the corresponding bit is setted to 1, to state that this region is dirty. Consulting the bitmap, a (periodic) checkpoint only saves the memory areas updated with respect to the last checkpoint. In turn the checkpoiting

policy will follow one of the aforementioned techniques to choose the interval χ.

Chapter 4

Moving to Hardware Based Recoverability

State recoverability is one of the most important mechanisms to be supported by speculative PDES platforms.

Even if simple in concept, state recoverability of the simulation objects rises enormous problems on the side of both performance and application transparency. In fact, the more efficient the recoverability support, the lower its overhead. On the other hand, application-transparent state restore typically demands more operations from an underlying recoverability layer, which further affects the tradeoff away from pure performance optimization.

Literature studies have jointly addressed performance and transparency aspects in state recoverability of simulation objects via multiple checkpointing techniques [10, 11] that, except for a few proposals based on (either conventional or non-conventional) hardware support [12, 13], rely on software implementations of the checkpointing support. Most of these proposals also entail overhead minimization techniques (e.g. via tuning of the parameters driving both checkpointing and, consequently, state recovery operations).

Nevertheless for the case of very fine grain simulation models, namely models based on events that require a few microseconds of CPU-time for being processed, the overhead can still represent an impairment to performance. A way to cope with this issue is the alternative recoverability technique based on reverse computing [14], where the forward application code is coupled with a (in some cases automatically generated [15]) reverse code version that is used to undo the state updates that are eventually revealed to be inconsistent. This solution pays-off especially in contexts where, beside having fine grain reverse (and hence forward) events, the portion of the state trajectory to be undone (*rollback length*) is short, which leads to a reduced number of reverse events to be processed per rollback operation.

Another aspect that plays a relevant role in case of speculative PDES with very fine grain models is the cost associated with cross-simulation-object scheduling of events, which may become predominant. This is typically achieved via message exchange (managed at the level of the underlying PDES platform), and, as discussed, the classical approach to undo the notification of an event that has been scheduled as a result of the processing of another event that is then detected to be non-consistent is to send a negative copy of it (the so called *anti-message*). Beyond potentially triggering a rollback operation at the recipient (in case the original copy of the message, namely event, was already processed) anti-messages lead to doubling the communication cost per-incorrect-scheduled events. Literature approaches have been proposed in order to reduce the number of message exchange operations, such as the ones based on message aggregation [16] or lazy-cancellation (lazy-antimessages) [17].

Despite the existence of a bunch of literature results on optimizing specu-

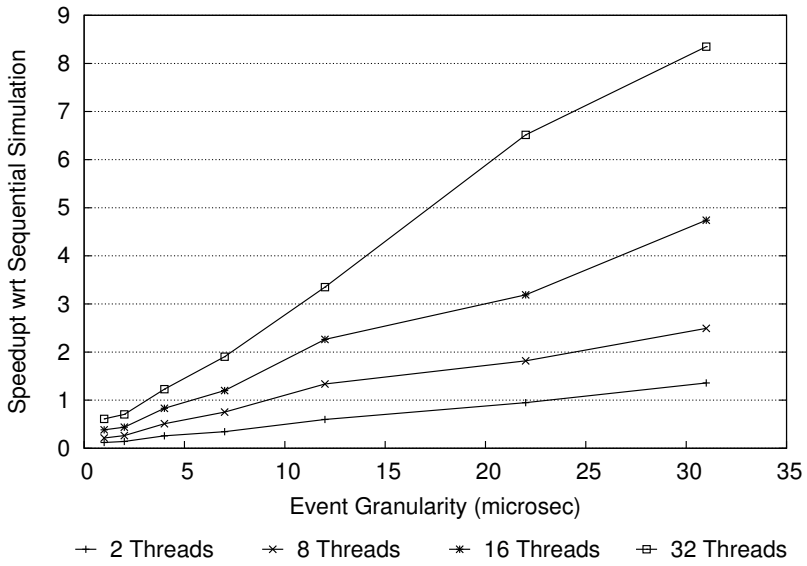

Fig. 4.1: Speedup for PHOLD while varying event granularity and number of threads

lative PDES systems, executing simulation models with very fine grain events on top of these systems in a performance-efficient manner is still a non-trivial achievement.

Just to provide some empirical evidence, in Figure 4.1 the speedup achievable by running the classical PHOLD benchmark for PDES systems [18] is reported (in a configuration with 2048 simulation objects) on top of the ROOT-Sim last generation speculative PDES platform ([1]) hosted on a 32-core off-the-shelf HP ProLiant machine, with respect to the sequential simulation of the same benchmark (same code) on a calendar-queue scheduler (still executed on the same machine). In the plot, the CPU-demand by PHOLD events is varied from a few to some tens of microseconds. The plotted curves show that speedup is unacceptable (it is a slow-down) for minimal CPU-requirements by the events, and is anyhow non-competitive (vs the em-

[1] https://github.com/HPDCS/ROOT-Sim

ployed number of threads) even when events last tens of microseconds.

In this thesis the issue of speculatively running PDES applications with very fine grain events efficiently on top of multi-core machines, is achieved by exploiting the Hardware-Transactional-Memory (HTM) support that is nowadays offered by off-the-shelf processors (such as the Intel Haswell). Overall, this approach is suited for contexts where conventional speculative PDES engines based on software recoverability (even the most advanced ones) fail to provide speedup just due to the excessively fine granularity of the simulation events (Figure 4.1).

4.1 Framing the HTM based proposal within literature

As discussed in the previous chapter, most of the state recoverability proposals in speculative PDES are based on software approaches. Since the classical software supports seems to completely fail in case of simulation models with very fine grain events, to search a most suitable technique the attention is turned to hardware solutions, in particular exploiting the transnational memory one.

In this section an overview of all the results studied to implement the recoverability relying on hardware support is provided. Some of these results will be used to support our proposal, while others will be shown to emphasize their cons and pros, underlining the different directions that have been take in this thesis.

First of all, a software proposal that exploits transactional memory is

described in [67]. Here the possibility to speculatively execute discrete event simulation applications in parallel on top of shared-memory multi-core systems by exploiting the TM paradigm for state recoverability is discusses. However, this work is still bound to software based recoverability, and does not attempt to exploit the innovative HTM-support provided by mainstream processor, as instead is done in our proposal. As a consequence, the path followed by [67] does not result particularly suited for very fine grain models, which is instead the scenario targeted by this thesis. However is possible to find some interesting checkpointing solutions oriented to speculative PDES based on hardware level facilities.

The really first hardware-based solution to state recoverability was published in 1992 [12]. This proposal introduces a special purpose device, called *rollback chip*. This device is used in this approach to keep the live state of the simulation object and gives the capability to perform the restore of past values. Then this solution require an additional hardware that is not part of the common hardware. Compared to this scheme, our proposal does not require specialized hardware, rather it is based on off-the-shelf general purpose HTM facilities.

The work described in [13] proposes an alternative way to use hardware support for state recoverability in speculative PDES. In this solution the checkpointing operation performed to copy the current state in the checkpoint buffer is realized in non-blocking manner via software managed DMA engines. In our proposal is preserved the same non-blocking advantage. The before image of the simulation object state is implicitly guaranteed to be available even when the simulation object is CPU-dispatched, thanks to the

underlying hardware transactional cache used to host the after image associated with event processing. In fact, the original memory remains untouched, while the updates are performed in the just mentioned transactional cache. On the other hand, in our proposal to restore the before image does not require software intervention, such that that aborting the transaction associated with the incorrectly processed event all the updates stored in cache are eliminated and leaving unchanged the original state. Differently the approach in [13], relies on software modules used to access the log and copy back the snapshot to be restored into the live state region.

Others hardware supported synchronization in PDES have been studied in [68, 69]. Both these proposals exploit hardware level facilities to determine the commit (or committable) horizon of a parallel PDES run, and hence to evaluate the safety of processed events (or of those to be still processed). These solutions are orthogonal to the one described in this thesis, since they do not use hardware level facilities to guarantee recoverability of the simulation model state trajectory. Moreover, the proposal in [68] stands as a theoretical design, given that it relies on an hardware component, that implement the reduction across threads that calculates the commit horizon, that has not been physically realized, rather it has only been evaluated via simulation. Instead, our engine is based on real off-the-shelf hardware facilities.

The recent proposal studied in [70] tackles the issue of exploiting the HTM support for speculative PDES. However, in this proposal the facilities offered by HTM are used as means for the atomic management of the event pool by the concurrent threads operating within the PDES platform. In particular in this approach the HTM-based transactions are used to encapsulate the con-

current accesses to the event pools. Rather, this thesis exploit HTM as the means for state recoverability while processing the events, which is achieved via the innovative algorithms provide, where ad-hoc data structures for determining the safety of speculatively processed events are managed within the transactional context.

As hinted before, approaches explicitly tailored to fine grain speculative PDES include those oriented to reduce the communication overhead, thus ultimately improving the computation to communication ratio. In [17] it is proposed a lazy-cancellation schema where an anti-message is sent out only after the assurance that the corresponding message to be cancelled would never been recreated. In [16] it is studied the message aggregation approach where messages are sent after batching them so as to amortize the send-setup cost. In [71] it is described the zero-copy message passing where the number of data copies along the path from source to destination is reduced to a minimum. In [72] it is presented the risk free synchronization schema where produced messages (events) are sent out only after having determined the consistency of the corresponding source event. Compared to these approaches, our proposal implicitly pursues similar objectives given that in our solution only committed output events (those produced by a committed transaction, namely a committed event execution) are actually flushed to memory. Hence is not allowed any non-committed newly produced output to live out of the hardware transactional cache, thus do not require sending anti-messages, given that only the output by committed events is reflected into memory.

Finally, compared to the reverse computing approach [14], which can

be considered as a means to reduce the state recoverability cost in case of fine grain events (where software based checkpointing would induce excessive CPU-time/memory overhead), our approach provides a different tradeoff given that our HTM-based speculative PDES engine guarantees state consistency with no intervention by the software. In fact our proposal do not rely on reverse events, rather on squashing the hardware transactional cache in the underlying HTM system, which leads the latency of the state restore operation to be independent of the length of rollback (as instead it does not occur in reverse computing schemes). On the other hand, our proposal allows for a speculative trajectory that has a number of speculative (uncommitted) steps bounded by the number of CPU-cores (namely the number of HTM caches available in the system), while reverse computing can be employed in contexts where the speculative chain of processed events does not undergo any specific constraint. In other words, in the approach described in this thesis optimism is limited by the available hardware resources, in terms of HTM caches, which is not the case for reverse computing.

4.2 Hardware Transactional Memory (HTM)

Before explaining the engine of this HTM-based approach to speculative PDES, can be useful to take a quick overview of the basic tools. Transactional memories are created with the aim to simplify concurrent accesses to shared variables, allowing a set of load/store instructions to be executed in atomic way. Even if this behaviour can be obtained by using locking primitives, the use of transactions can provide a simplest and efficient way to manage concurrency. The goal of a transactional memory system is to transparently support the definition of portion of code considered a transaction, providing

atomicity, consistency and isolation.

Nowadays transactional memories are available via software and hardware implementations. A *Software Transactional Memory* (STM) provides transactional memory semantics embedded in a runtime library, and requires minimal hardware support (e.g. `compare and swap` (CAS)) available on all modern architectures. While this approach give the possibility to use transactional memory on common machines, software implementation usually implies performance penalty, especially compared with hardware one.

Hardware Transactional Memory (HTM) systems support the execution of transactional portion of code relying on hardware support given by the processor. The `compare and swap` (CAS) instruction hinted before, or still the `load-link/store-condition` (LL/SC) one, can be viewed as basic transactional memory support, however, designed to manage the size of a single native machine word.

During the experimentation phase of this thesis we have used a machine equipped with two Intel Haswell processors, that is the first mainstream architecture to include hardware support for transactional memory.

From a user side, to use the transactional memory support on a HTM-equipped machine, is enough to encapsulate the portion of code to be executed in transactional way:

```
1 retry:
2     if (_xbegin() == 0) {
3         /** critical section */
4         _xend();
5     }
6     else {
7         goto retry;
8     }
```

`_xbegin()` is the instruction used to mark the begin of the transactional portion of code, while `_xend()` is used to end the transaction trying to commit it. A transaction execution might fails for different reason during its execution (e.g. cache full, nested transaction, or also by using `_xabort()` instruction) or might abort during the commit phase, if the transaction itself produces conflict in memory. In any case, if the transaction go in abort, the execution flow is rolled backward until the begin of the transaction, returning this time an error value. In this way, if the transaction go in abort, the application writer can choose what to do, e.g. retrying to execute the transaction.

On the other side, during a transaction, the hardware keeps track of which cache lines have been read from and which have been written to. When the program try to commit the transaction, the hardware checks if the memory portion read or written have been modified and if so goes in abort, otherwise commits the transaction and, in atomic way, writes data in memory. Thus, if two transactions concurrently read the same memory portion, there is no problem. Otherwise if at least one of the two transactions modifies a memory portion read from the others, there is a conflict and one of the transaction will abort (based on the commit time). This means that two thread are able to concurrently work on the same memory region and both the transactions will commit unless of read/write or write/write operations on the same data. Moreover, if a conflict occurs, at least one of the two transaction will commit.

In the end, the use of hardware transactional memories to support rollback operation in speculative PDES, seems to be extremely natural since the optimistic concurrency, namely the speculation, is the basic mechanism used to implements transaction.

4.3 HTM-based Speculative PDES

As hinted, this thesis describes an HTM-based engine developed to efficiently perform recoverability in speculatively running PDES applications with very fine grain events on top of multi-core machines.

In this approach the simulation kernel is slightly different due the nature itself of the problem and the hardware exploited.

First of all this solution is developed to work on a single multi-core machine due to the size of events and the transactional support. In fact, in this engine, the optimism in the speculation is constrained by the number of (real) cores available on the machine since each core is able to support only one transaction per time. This happens for two reasons: first, if a thread is descheduled to left the core to another one (then if the engine uses more threads per core), the context switch should invalidate the cache, aborting the transaction; second, if one thread tries to execute more events before commits, the cache could fill, aborting the transaction. Thus, since the execution in the future are limited to a finite number of events, and since the execution time of very fine grain events is negligible compared to network delays, introduce remote communication with other simulation kernel instances would entail an heavy decrease of performance.

Moreover, since the events that can be executed speculatively are constrained in number, in this solution we have decided to not tie an LP to a specific thread. In this way each thread will execute the event not yet performed with the smallest timestamp in all the system, regardless the recipient LP. This gives us the possibility to think the pending event queue with a different point of view.

Additionally, since each thread can manage only one event per time, this carries on the event until its commit, removing the necessity to keep a queue

for the processed events.

In the end, all these aspects, linked to the underlying hardware support, give the possibility to develop a lighter and dynamic simulation kernel, that is what we need to manage very fine grain events.

4.3.1 Structures

In the classical PDES approach, as many pending event queues are used as there are the number of LPs used by the simulation. In our approach the schema to manage the events to be processed has be redesigned to reflect the lightweight and dynamic of the models addressed. Since the LPs are no more tied to a single thread, we consider a scenario where all the events that have been scheduled, including the simulation startup events, destined to whichever simulation object, are kept together in a single pool. In particular, the structure used to keep events to be processed is a modification of the classical *calendar-queue* [36].

This structure represents the behaviour of a typical cyclic desk calendar. One schedules an event by simply writing it on the appropriate page with the relative time and year[2]. The time at which the event is scheduled represents the priority, while the year says if the event has to be processed now or at the next turn. In its implementation each page of the calendar queue is an ordered linked list and an array containing one entry for each day of the year is used to point the relative list. The struct keeps the information about the current entry of the array. When is performed a dequeue from the calendar queue, one check the year of the first event in the list: if it is the current one,

[2]In the real implementation there is only the information about the timestamp of the event and an additional information in the structure that say which is the limit of time to be considered in the current year.

the event is taken, otherwise skips to the next entry of the array. When is reached the end of the array, the research starts again from the first entry. The length of the array is chosen to be long enough that most event will be scheduled for no more that a year, then the length of a year and of a day are adjusted if this condition is no more in place.

The calendar queue seems to be good for our purpose since, to insert an element is enough to point the right entry of the array (with a module operation) and search the correct position in the list, then with a complexity linear in the size of the day, while to get an event in most of the cases is enough to take the first element of the current day, providing average constant-time $O(1)$ performance.

However, the classical calendar queue is constrained by the fact that the new events inserted in the queue, as in a real calendar, have to be associated with a timestamp greater than the last event dequeued. In our solution this is not acceptable since, due the concurrence, can happen that is inserted an event with a timestamp smaller than the last extracted. Let's consider a quick example: a first thread gets an event e_1 with timestamp T_{e_1}, while a second thread gets the next event e_2 with timestamp T_{e_1} greater than T_{e_1}. At this point, the execution of e_1 might generate an event e_3 with timestamp T_{e_3} such that $T_{e_3} < T_{e_3} \leq T_{e_3}$ that violates the constraint imposed by the classical calendar-queue. Thus, the classical implementation of the calendar-queue is modified in order to allow it to be filled with an event e whose timestamp $T(e)$ is such that $T(e) < T_{min}$, where T_{min} represents the minimum timestamp of all the events. This is done by simply moving the current day to the one of the event inserted out of order, and recalculating the period of the year.

In this engine, the sorting of the elements into the calendar-queue is based on event timestamps, but the event records also entails information

determining what simulation object is the target of an event (since we have a single pending queue for all the system), and the actual event type/payload, as typical of PDES environments.

All the events kept in the calendar-queue are *schedule-committed*, with the meaning that they will never be retracted (e.g. via negative copies). In fact, in this approach events scheduled during the execution of an HTM-supported event are flushed to the calendar-queue only if the transaction associated with the processing of the event successfully commits. This, in its turn, implies that the event is no longer rollbackable, therefore the output events it produced will never be retracted.

This implies also the need for a private pool for each thread to keep the output event generated during the execution of the event. This pool normally is constrained by a very small size. In fact, for discrete event models that reach steady state behavior while being processed, it is classical to have an average number of newly produced events per any individual event execution which is of the order of one unit. If this behaviour were unattended, we would experience an unbounded growth of memory requested for keeping scheduled but not yet executed events while the run is in progress.

We target the scenario where the maximum number of worker-threads employed in the speculative platform is upper bounded by the number of available CPU-cores, say N. This is a classical configuration avoiding interference by deschedule/reschedule operations in parallel applications[73, 74, 75] at least for cases where the platform is temporarily dedicated to a specific application.

The calendar-queue data structure is coupled with an array of N entries, named `processing[]`. This structures is the core of all the synchronization mechanism since it is used to decide when an event can be considered *safe*.

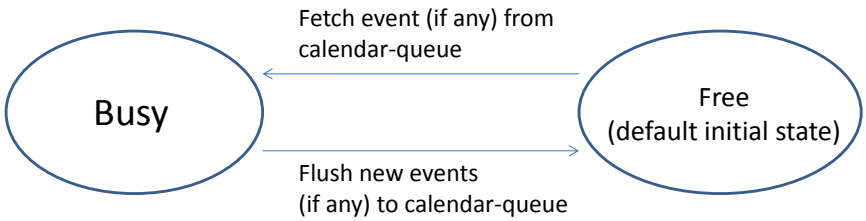

Fig. 4.2: State diagram for the generic worker-thread WT_i

In other words, this structure is used to compute at each time the GVT value. Differently by the common solution, this operation can be performed quickly thanks to the organization of the whole engine. The i-th entry of this arrays is used to keep data, say a timestamp value, related to the status of the i-th worker-thread operating within the speculative PDES platform, which we denote as WT_i.

At the startup of the simulation this array is initialized in such a way to keep in all the entries the special value ∞. This value means that the relative thread has not event to process. At the same time, the calendar-queue is initialized in such a way to keep the initial events that, depending on the simulation model configuration, will fire any initial state transition in the model execution on whichever simulation object.

4.3.2 Basic Engine

Now is possible to go on to discuss the behaviour of the engine during its execution. During the simulation, each worker-thread WT_i lives in the state diagram shown in Figure 4.2.

Algorithm 2 Fetch operation - worker-thread WT_i

```
1: procedure FETCH [ATOMIC]RETURNS: event
2:     e ← GetMinimumTimestampEventFromCalendarQueue
3:     if e = NULL then
4:         processing[i] ← ∞
5:     else
6:         processing[i] ← T(e)
7:     end if
8:     return e
9: end procedure
```

At the startup each thread is in the *free* state. This means that, when the system is started, all the threads still have not pending simulation event (to be executed) yet assigned.

When a thread is in the *free* state, it tries to get an event to process by asking to the calendar-queue the next event to process. Thus, WT_i leaves the *free* state and enters the *busy* one upon performing a FETCH operation that leads to the actual extraction of some event to be processed from the calendar-queue. The FETCH operation executes the actions described in Algorithm 2.

In particular, first, it atomically try to extract the event e with minimum timestamp that is currently registered into the calendar-queue (if any), and records the extracted timestamp value into the entry `processing[i]` associated with WT_i. Atomicity is necessary, since it avoids that two different worker-threads take care of processing the same pending event. Moreover, if two worker-threads, say WT_i and WT_j execute the FETCH operation concurrently, and the two operations are serialized in such a way that the two threads extract from the calendar-queue, respectively, the event e and then the event e', it is guaranteed that $T(e) < T(e')$ ([3]). Hence it is also guaran-

[3]Here we implicitly assume that no simultaneous events will ever exist. However, the case of simultaneous events, where $T(e)$ may be equal to $T(e')$, will be explicitly dealt

teed that `processing[i]<processing[j]`, given that the two array entries are updated according to the established serialization order.

It is important to note that the FETCH operation can be executed in constant-time average performance, since the calendar-queue guarantees $O(1)$ average-performance. Hence, the usage of a conventional spin-lock to achieve atomicity of the FETCH operation should not represent a scalability impairment, at least at the CPU-core count that currently characterizes processors offering HTM support ([4]).

If no actual event is extracted from the calendar-queue by WT_i while executing the FETCH procedure (i.e. the calendar-queue is found to be empty upon being accessed), the entry `processing[i]` is set to the default initial value ∞. In the main loop of simulation processing, this scenario will simply lead to retry the FETCH operation, leaving the worker-thread in the free state, given that no job to perform has been assigned to it. Moreover, setting `processing[i]` to ∞, the worker thread WT_i notifies to others that it has no event to process and there are not other event to be processed in the future, then it can be ignored in the computation of the safety to commit an executed event.

When WT_i has reached the end of the execution of the current event(that happens in the *busy* state), tries to commit it. If the commit is allowed, the thread flushes any speculative operation (e.g. memory update) associated with event processing, which it kept at the level of the HTM cache, is allowed to be flushed to memory for making it visible. In fact, as hinted before, during the transactional execution of an event, the update in memory are not

with later in the article.

[4]For scaled-up CPU-core counts, as we may expect it will be the case for next-generation HTM-equipped machines, we can envisage the reliance on wait-free algorithms rather than lock-based ones [76, 58].

Algorithm 3 Flush operation - worker-thread WT_i

```
1: procedure FLUSH [ATOMIC](event_set E)
2:     ∀e ∈ E: InsertInCalendarQueue(e)
3: end procedure
```

visible, but are done in a private cache, with the aim to copy it in memory only when the transaction reaches correctly the end. However, has hinted before, the newly scheduled events (possibly produced by the current event execution) produced during the execution of the HTM-based transaction, are stored into a thread-private buffer, such that the buffer content is made visible only upon committing the HTM-based transaction. Hence, right after committing the event processing phase, these events can be flushed (outside of the transactional code-block) from the thread-private buffer to the calendar-queue, making these available to be processed.

The set of all the activities carried out by any worker-thread WT_i right after the commit phase, is referred as FLUSH procedure. This phase is depicted in Algorithm 3. As is possible to see, also this operation is executed in atomic way, in particular relying on the same spin-lock used in the FETCH operation, and simply inserts all the newly produced events (if any) into the calendar-queue. As an additional note, the FLUSH operation may be requested to atomically insert into the calendar-queue more than one event, but, as seen before, a well designed model produce an average of at most one new event. As a results, in most of the scenarios we can expect constant-time average performance also for the FLUSH operation.

Upon executing this procedure, the worker-thread switches back to the *free* state.

At this point the flushed events will be eventually extracted from the calendar-queue by any thread that will (re)transit into the *free* state, via the execution of FETCH operations. As seen before, a new event e' that is

flushed to the calendar-queue might be associated with a timestamp $T(e')$ such that $T(e') < T(e)$, where e is an event previously fetched by some worker-thread WT_j (namely, the one with minimum timestamp across those stored in the calendar-queue at the time of the FETCH operation). In this case, the newly scheduled event e' stands in the past of e, which plays a role in the determination of event safety (causal consistency) within the speculative processing scheme.

As hinted, the array `processing[]` is a pivot structure to guarantee consistency in the simulation. In fact, the values registered in the entries of this array are used to define the order according which the events currently handled by worker-threads are committed. In particular, they establish the order according to which the HTM-based transactions implementing the processing of the events need to be committed. Hence, the entries of this arrays are used in a manner similar (at least in spirit) to what is done by Lamport's bakery algorithm [77], which is classically used for defining the order according to which concurrent threads can access a same critical section.

In this approach the condition that tells whether a worker-thread WT_i can safely commit the event it is handling is expressed as:

$$\forall j \neq i : \texttt{processing[i]} < \texttt{processing[j]}$$

This condition indicates that the (possibly speculatively) executed event is associated with the current lower bound timestamp across all the not yet (or just) processed events in the system. Thus, if the condition is verified, this means that the timestamp of this event represents the commit horizon, and then the event can be safely executed or (in case of already carried out speculative execution) safely committed. In other words, if a timestamp Ts verify the above condition, it means that Ts is the current GVT. The pseudo-code implementing the safety-check is provided in Algorithm 4.

Algorithm 4 Safety-check - worker-thread WT_i

```
1: procedure SAFE RETURNS: BOOLEAN
2:     T̂ ← MIN_{∀j≠i}(processing[j])
3:     if (processing[i] < T̂) then
4:         return TRUE
5:     else
6:         return FALSE
7:     end if
8: end procedure
```

As the reader may observe, the SAFE procedure does not require to be executed atomically. This is due to the fact that when any worker-thread WT_j executes the FLUSH procedure, it leaves `processing[j]` untouched. Could happen that some values are modified during the execution but, for the natural ascending order given by the execution of subsequent events, the new value will be ever greater than the minimum between all the events.

Then, starting from the above algorithms is possible to construct the basic engine of the simulation kernel. The execution loop of any worker-thread WT_i is the one reported in Algorithm 5.

The first operation of the main-loop is the FETCH, performed to get the event with the smallest timestamp from the pending queue. This operation in retried until an event is got.

Right after is carried out the check-safety to choose the way (speculative or not) in which the event e currently assigned to WT_i needs to be processed. If the safety-check at line 7 is verified, then the event can be executed with no need to start an HTM-based transaction for recoverability purposes, since the effects produced by its execution are safe and no other thread can invalidate them. In other words, the timestamp of the event is already known (prior to the actual processing of the event) to correspond to the commit horizon of the speculative run. Otherwise, if the safety-check at line 7 is not verified, WT_i

Algorithm 5 Main loop

```
 1: procedure MainLoop
 2:     while ¬endSimulation do
 3:         e ← FETCH( )
 4:         if e = NULL then
 5:             retry from line 3
 6:         end if
 7:         if Safe then
 8:             event_set New_events ← ProcessEvent(e)
 9:             FLUSH(New_events)
10:         else
11:             BeginTransaction( )
12:             event_set New_events ← ProcessEvent(e)
13:             if Safe then
14:                 CommitTransaction( )
15:                 FLUSH(New_events)
16:             else
17:                 AbortTransaction( )
18:                 retry from line 7
19:             end if
20:         end if
21:     end while
22: end procedure
```

processes the event speculatively within an HTM-based transaction (hence in a recoverable manner, given that the transaction can be aborted, if needed).

If the event is executed speculatively, after the processing phase, the safety-check is re-executed. Now, if the event has become committable (namely, it now lies on the hopefully advanced commit horizon) then the actual commit takes place, with installation of the event side effects that become visible. In the negative case, the whole process of safety-assessment and safe vs speculative execution is retried, resuming from line 7, right after the fetch operation. Unfortunately, if the check-safety in line 13 fails, the transaction has to be aborted and the event re-executed. In fact, is check-safety fails, this means that the thread has read at least an entry of the array

with a value smaller than its current time and now, if the thread wants to retry the control, to be safe should read some different values in the array, which would fail the transaction, due a read/write conflict.

When a worker thread complete correctly the execution of an event resulting safe in one of the two controls, is performed the FLUSH operation that injects the new produced events in the system and switches back the thread to the *free* state.

In the end, the structure of Algorithm 5 leads any worker-thread WT_i to be able to execute its currently assigned event in safe mode or in speculative mode. In the latter case, we have chances that, at the end of the speculative processing phase, the event has became committable, thanks to the previous commit of events that stand in its past, and the absence of newly injected events still standing in its past.

4.3.3 Optimizations

This approach has to be able to manage events quickly, reflecting the nature of the models addressed. Moreover, since the consequences due to the performing the check-safety on an unsafe transaction, it is important to choose the the right moment to perform it. In this section will be presented some optimizations studied to take full advantage of the hardware provided.

4.3.3.1 Handling Simultaneous Events

The safety condition discussed above is based on having some worker-thread WT_i in the *busy* state that is in charge of processing (or has speculatively processed) the event with the current absolute minimum timestamp within the whole system. As hinted, this condition might never be verified with simultaneous events, namely events marked with the same identical times-

tamp. If simultaneity of events would be admitted in the simulation model, then Algorithm 5 would give rise to live-lock.

Guaranteeing progress in speculative PDES systems in the presence of simultaneous events is a well understood problem that has been extensively studied in literature [53], which is not specifically bound to this proposal. Hence different literature solutions for tie-breaking simultaneous events (see, e.g., [78]) can be exploited for integration with our HTM-based speculative PDES system. A baseline approach could consist in comparing (for safety assessment) both timestamp values and worker-thread identifiers, according to the philosophy underlying Lamport's bakery algorithm [77].

In this engine is used a baseline variant for safety-assessment (of the event currently handled by any worker-thread WT_i) in case the simulation model admits simultaneous events, which is based on the adoption of the following predicate:

$$\forall j \neq i : \texttt{processing[i]} \leq \texttt{processing[j]}\ \texttt{AND}\ i < j$$

This variant does not consider (possible) causal relations across simultaneous events, since the tie-break is exclusively based on worker-threads' identifiers. However, the achievement of liveness with simultaneous events, while jointly guaranteeing causality across them, could be reached in our scheme by relying on causal-timestamps (see, e.g., [79]). Hence the entries of the `processing[]` array could be simply setup to keep causal-timestamps rather than classical ones if a scenario with causal simultaneous events would need to be dealt with.

4.3.3.2 Non-zero Lookahead

From the PDES literature it is well known that the simulation model *lookahead* can play a important role on the efficiency of synchronization. The

lookahead is a value, assigned to the simulation model by the model write, that represents the minimum interval between two consecutive events. In other words, when the execution of an event e with timestamp T_e produce a new event e', this must have a timestamp $T_{e'}$ such that $T_{e'} \geq T_e + L$, where L is the lookahead assigned to the discrete model. Hence e will not give rise to causality dependencies up to time $T_e + L$. Overall, the lookahead is the ability to predict that nothing will occur in logical time up to a virtual time point that is a function of the current time.

Although it plays a major role for conservative PDES [2], since in this kind of approach this is the only way to allow the execution of concurrent events, it can play such a role also in speculative PDES systems. However, the traditional way the lookahead is used is to determine (a-priori for conservative PDES vs a-posteriori for speculative PDES) the safety of the events that are executed by a simulation object when assuming that the object is a sequential entity.

In this approach, objects are no longer sequential entities, given that two different worker-threads can contemporaneously reside in the *busy* state by having fetched two events destined to the same simulation object. Let us indicate with T_e and $T_{e'}$ the timestamps of these two events and assume, with no loss of generality, that $T_e < T_{e'}$. Suppose, still with no loss of generality, that the simulation model has lookahead L, such that $T_e + L > T_{e'}$ and then e will not generate event to be executed before e'. In such a scenario, we cannot assert that the event e' is safe (which might lead it to commit before e is committed), given that it may need to access the simulation object snapshot that has been produced by e. Overall, event safety calculation on the basis of the lookahead can be still adopted for scenarios where the event in the past, say e in our example, is associated with a simulation object

different from the one associated with the event in the future, say e'. In such a scenario, the condition $T(e) + L > T(e')$ leads to the fact any new event produced by the source simulation object, the one processing e, will not be causally related to e', thus also if e will produce an event to be processed on the same simulation object of e' this will occur only after e'.

Hence, the overall HTM-based speculative PDES engine organization can be modified to support the *lookahead*. This is done by including an additional array `destination[]`, with N entries, such that, in case WT_i is in the *busy* state, `destination[i]` keeps the identifier of the simulation object that is the target of the event to be processed by WT_i. This way we can distinguish a priori whether different events that are concurrently handled by two worker-threads operate on disjoint portions of the simulation model. By exploiting this new array, and the lookahead value L (if any), the safety of the event to be processed by WT_i can be assessed according to the following condition:

$$\begin{aligned}
&\forall j \neq i \texttt{ such that destination[j]} \neq \texttt{destination[i]} : \\
&\qquad \texttt{processing[i]} < \texttt{processing[j]} + L \\
&\texttt{AND} \\
&\forall j \neq i \texttt{ such that destination[j]} = \texttt{destination[i]} : \\
&\qquad \texttt{processing[i]} < \texttt{processing[j]}
\end{aligned}$$

The above logic can increase concurrency with a double effect. First, it allows the concurrent execution of more than one thread in safe mode. Second, allows the speculatively processed events (via HTM-based support) to be committed in non-strictly increasing values of their timestamps, as instead it needs to occur when relying on the condition adopted by the SAFE procedure in Algorithm 4. Further, the above predicate can be still integrated with the aforementioned logic for managing simultaneous events based on

worker-thread identifiers. As a last note, such a predicate is fully independent of the actual interaction graph across the simulation objects. This is a choice aligned with the classical objectives of speculative PDES synchronization, which does not require a-priori knowledge of the (potential) interactions across the concurrent simulation objects.

4.3.3.3 Throttling

Another optimization introduced in this section is related to line 13 of Algorithm, where the SAFE procedure is called while being in HTM-based transactional contexts. As hinted, it is not useful to call the check-safety control multiple times inside the same transaction while handling event e, thus following a polling approach for safety-assessment. This is because any update occurring onto the array checked by SAFE leads to the abort of the HTM-based transaction associated with the processing of e in case the safety-check was previously invoked within the same transaction and the array was updated via FETCH operations by concurrent worker-threads. For this reason, if the SAFE control fails the first time, the algorithm forces it to abort, without reach the end. Then, the aim is to make the SAFE control when it is more likely to return true. Thus, in order to increase the likelihood that, in case e has become a safe event along its speculative processing interval, we can actually detect its safety upon calling the SAFE procedure in line 13 of Algorithm 5, it is devised a throttling approach (which is a classical technique for reducing the likelihood of rollbacks in speculative PDES, see, e.g. [51]).

In particular, the procedure SAFE is modified to return to the invoking worker-thread WT_i currently handling event e, the number of entries of the `processing[]` array which keep timestamps that are lower than the

timestamp kept by `processing[i]`. Denoted with K this number, which corresponds to the minimum number of events that need to be committed before a speculative run of the event e bound to WT_i can be committed. It is the minimum, not the exact value, because K does not account for events with timestamps lower than `processing[i]`, if any, which might have been already inserted into the calendar-queue, but that might have not been fetched for processing.

Clearly, when calling SAFE in line 7 of Algorithm 5, K is zero if the event bound to WT_i is safe, while K is different from zero in case the event is not detected to be safe. However, in case the event e is not yet safe when calling the SAFE procedure in line 7 of Algorithm 5, K provides an indication of the minimum number of events from which e may causally depend, which need therefore to be committed before committing any speculative run of e.

Thus, to improve the Algorithm 5 with the throttling schema, is inserted a delay before the SAFE procedure. This delay is implemented as CPU-busy loop, since operating system `sleep` cannot be used, given that it would lead to aborting HTM-based transactions because of a mode-change along thread execution.

In particular, the throttling delay is computed as:

$$\delta \times K \times \alpha \tag{4.1}$$

where:

- δ is the average event granularity for the executed simulation model;
- α is a parameter falling in the interval [0,1], depending on the current execution.

The parameter α is a variable introduced to follow dynamically the system behaviour, since is determined following a classical hill-climbing approach,

	Speculative engine type	
	classical	HTM-based
suitability for very fine-grain events	no (or limited)	yes
unbounded chain of speculative events	yes	no
intra-simulation-object parallelism	no	yes
suitability for very large scale platforms	generally yes	no

Table 4.1: Summary of HTM-based vs classical speculative PDES.

similar to the one adopted in [80] for tuning the checkpoint interval to the value that optimizes performance. In this hill-climbing approach the metric used to dynamically set α is the number of committed events per wall-clock-time unit. In fact, if on one side delaying the SAFE control increases the likelihood of success, on the other hand, increasing too much this interval is possible to reduce the benefit given by the parallel execution. Moreover, if the transaction is too long, is also increased the probability of abort due to the arrival of an interrupt. In other words, throttling is regulated in such a way to increase the likelihood of performing useful work while the worker-threads reside in the *busy* state. Is also included an ϵ-greedy scheme to avoid stalling in local maxima.

4.4 HTM-based vs Classical Speculative PDES: a Summary

In this section is provided a comparison of the main differences (as evaluated by relying on four reference indices) between our HTM-based approach and classical speculative PDES, as shown in Table 4.1.

The latter allows for (ideally) unbounded chains of speculatively processed events along the execution path of each individual simulation object

(it may only depend on memory limits for keeping speculatively executed event-buffers), while the HTM-based approach allows for up to N (number of CPU-cores) speculative events to stand out, given that recoverability relies on the hardware transactional cache. However, this seems to not be a big problem since, using a single pending queue, all the event in processing are "near" to the commit horizon and then they can be committed in a short time, allowing the threads to get a new event to process.

On the other hand, the HTM-based approach allows intra-simulation-object concurrency while classical speculative PDES does not allow for it. In fact, the hardware transactional memory automatically resolves data conflicts arising while processing in parallel events destined to the same simulation object (this leads to squash/retry of the corresponding HTM-based transactions in case some conflict materializes). This means, after all, that even with a smaller number of simulation objects, the performance would remain similar, differently by the classical approach where, due to the exclusive use of the simulation object, the performance are strictly related to the size of LPs set.

As for usefulness when parallelizing very fine grain models, the HTM-based approach fully fits it, while the traditional approach may provide limited usefulness. On the other hand, the classical approach has been already shown to scale to very large computing platforms (see, e.g., [4]), while our solution is intrinsically targeted at limited scale machines (e.g. because of the atomicity requested in manipulating the shared calendar-queue across the worker-threads, or the shared arrays of meta-data).

However, at current date, machines with HTM support show relatively limited number of cores. Hence our approach looks suited for current HTM platforms. Also, for very fine grain models, significant reduction of the completion time can be achieved even with limited (but well exploited) amounts

of CPU-cores, which is one target we achieve, as shown by the experimental data provide in the following section.

Concluding, as hinted in Section 4.1, in literature there are not valid approaches to carry on parallel simulation of fine-grain discrete event, and this HTM-based approach represents a valid solution as proved by the experimentation shown in the next section.

Chapter 5

Experimental evaluation

This chapter is dedicated to the analysis of our approach. The experimentations carried aim to evaluate the benefit given by this approach in different contexts. In particular the tests are done on two different models, varying the requirements by the events, both in terms of duration and memory usage.

5.1 Hardware Setup

The reference computing platform is an HTM-equipped machine which entails two 4-core Intel(R) Xeon(R) CPU E3-1275 v3 3.5 GHz processors with hyper-threading support and 24 GB RAM. During the tests we have not used the hyper-threading (hence the whole available set of 8 cores) in order to avoid interference by different threads on transactional-cache portions that are shared across hyper-threaded cores. In fact, during the tests we noticed that, going over 4 threads (i.e. 4 physical disjointed cores, with separate caches), the ratio of occurrences of abort due to cache full has increased sharply, leading HTM-based speculative processing of events to abort due to phenomena that are not directly imputable to our speculative processing

support. This hardware has been used only to perform simulation during all the tests, then the results reflect a scenario where the machine is completely dedicated to high performance simulation.

The Operating System used by the machine is 64-bit *Ubuntu* 12.04.2 LTS, with *Linux* Kernel version 3.5.0-23. The linking and compiling tool used is *gcc* 4.8.1.

5.2 Benchmark Applications

PHOLD Is a model used to benchmarking simulation kernels with the possibility to vary the duration of events. In this benchmark, each simulation object executes sham events with the only purpose of advancing the local simulation clock to the event timestamp [81]. Each time an event is executed, a new sham event is scheduled, destined to whatever object inside the simulation, with the timestamp incremented following an exponential distribution. The execution of an event has included a busy loop (which emulates a specific CPU time for event processing, and hence a specific event granularity). In our case, varying the number of iterations of the loop we can benchmark our new approach with very fine grain events.

Terrain-Covering Ant Robots (TCAR) This is an exploration and mapping agent based simulation model developed on the basis of the results in [82]. Specifically, in this model a group of agents (i.e. *ant robots*) is set out into an unknown space, with the goal to fully exploring it, while acquiring data from sensors (e.g., cameras, lasers, ...) which are used to map the environment. 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 *pheromones count*: each subregion is assigned a counter which

gets incremented whenever any robot visits it, i.e. tracks the number of *pheromones* left by ants, to notify other ones of their transit. To decide what direction to take, ant robots adopt a greedy approach, so that when a robot is in a particular subregion, it targets the neighbour with the minimum trail count. A random choice takes place if multiple subregions have the same (minimum) trail count. The terrain is represented by an undirected graph, then any agent is able to go from one cell to another one. This mapping is done by representing the terrain as a matrix composed by hexagonal cells. At the beginning of the simulation, the agents are disposed on two opposed border spots. In our implementation of this model, each simulation object models an hexagonal subregion of an overall region to be explored.

5.3 Results

We initially tested our HTM-based speculative PDES system by relying on PHOLD benchmark described in Section 5.2. We included 2048 simulation objects in the simulation model, each one scheduling events for itself or for the other objects. Specifically, upon processing an event, the probability to schedule a new event destined to another simulation object has been set to 0.2, which is representative of scenarios with non-minimal interactions across the different objects. Also, the initial population of events has been set to 1 event per simulation object, while the timestamp increment determining the actual timestamp of newly scheduled events has been set to follow the exponential distribution with mean value equal to one simulation time unit. The model lookahead has been set to a minimal value computed as the 0.5% of the average timestamp increment. For this benchmark configuration, we varied the CPU-demand for processing the events in the interval between

2 and 12 microseconds, which has been done by appropriately setting the classical busy-loop characterizing PHOLD event processing steps. Hence we studied the system behavior when moving from very fine to fine grain event configurations. We have run this benchmark by varying the number of employed threads from 1 to the maximum number of physical CPU-cores, say 4, in the underlying HTM-equipped machine. For the case of single-thread runs, the execution time values are those achieved by simply running the application code on top of the calendar-queue scheduler, while for all the other settings of the number of threads we relied on the HTM-based parallel implementation we presented ([1]). We have also run the same identical model by relying on the last generation ROOT-Sim speculative PDES engine, which offers a pure software-based support for recoverability.

The results obtained are shown in Figure 5.1 where is reported the observed speedup values versus the single-thread execution time. Each reported value is the result of the average over 5 different samples. The data clearly show how our HTM-based proposal definitely outperforms the traditional style PDES engine. Particularly, our solution allows achieving speedup that ranges between 1.5 and 3.6, with the highest values achieved when the granularity of the events increases towards 12 microseconds. Also, with 4 threads it provides speedup above 3 as soon as the event granularity is of at least 4 microseconds.

Instead, the traditional PDES engine only provides slow-down, which again confirms the unsuitability of the classical software-based recoverability support for speculative execution of models with very reduced event granularity.

To complement the above data, we report in Figure 5.2 the number of

[1]This is available as open source at https://github.com/HPDCS/htmPDES.

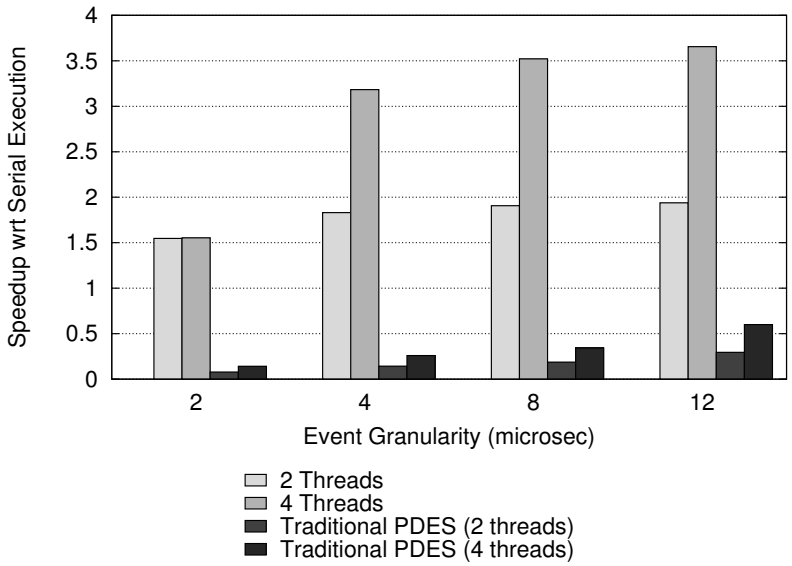

Fig. 5.1: Speedup values for PHOLD

transaction aborts due to negative safety-check of speculatively processed events. By the data we can see how the number of aborts tends to decrease while increasing the event granularity. On the other hand, the speedup observed for lower event granularity values is higher than the one observed for coarser grain events. This is a reflection of the fact that with very fine grain events, a higher number of event processing retries pay-off, which is essentially due to the fact that recoverability tasks are extremely light, thanks to the HTM-based support. On the other hand, recoverability is still light with coarser grain events, but the retry of coarser granularity events does not favor performance to the same extent we get for the case of finer grain ones.

In Figure 5.3 we report an additional set of data showing how the HTM-support for event speculation influences the execution dynamics. In particular, we show the probability that some event gets eventually committed after having been executed speculatively within an HTM-based transaction. The

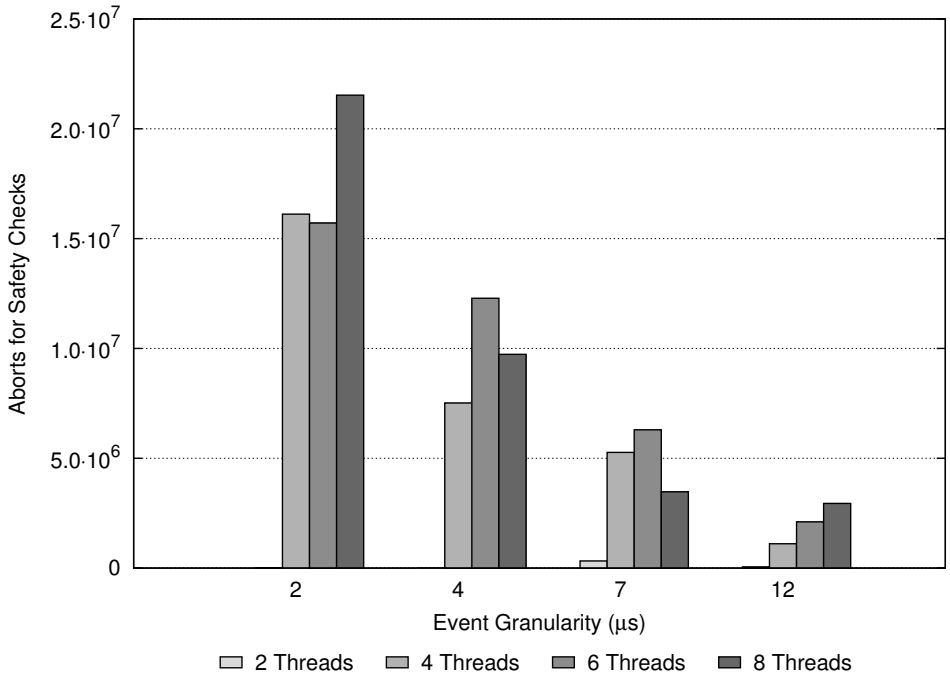

Fig. 5.2: Aborted transactions

data show an interesting trend where for lower levels of parallelism (say 2 threads) the HTM support does not influence speculation (and its performance effects) significantly. In fact, the likelihood for a committed event to have been executed within an HTM-based transaction is very low, indicating how the most important contribution to parallelism in the execution is provided by the exploitation of the lookahead, which allows for processing events in a safe mode outside any transaction. On the other hand, when increasing the level of parallelism, the HTM-based support starts to play a relevant role, given that the probability for a committed event to have been processed speculatively within some HTM-based transaction increases up to (slightly less than) 0.2. Also, the clear increase of the usefulness of HTM-based speculative processing when moving from 2 to 4 threads indicates a potential for scalability of our approach to HTM-equipped machines with

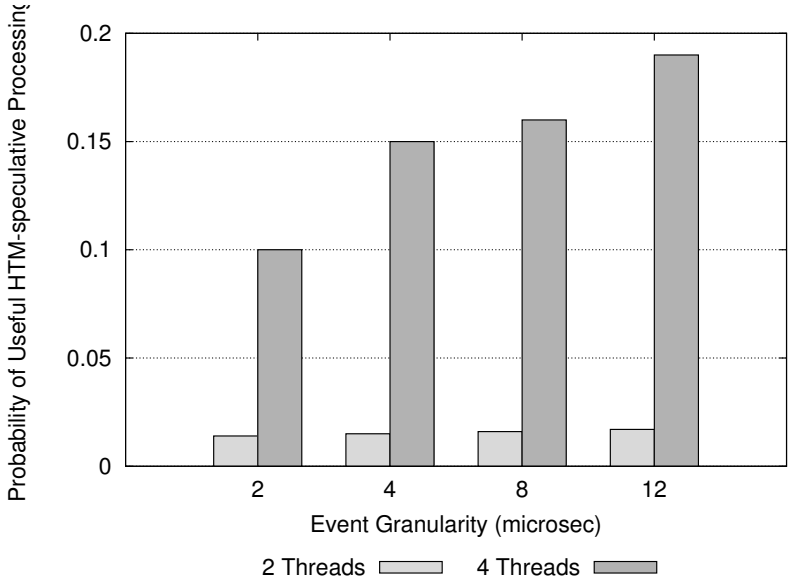

Fig. 5.3: Usefulness of HTM-based speculation

larger numbers of physical cores.

Interesting results are given by the comparison between the speedup obtained using the throttling mechanism versus the results obtained by disabling it. In particular, the throttling mechanism allow speedup improvements that grow up to about 20% (see the configurations with 2 and 4 microseconds event granularity). Also, the gain by throtling appears when the number of concurrent threads is grater than 4 (say 6 or 8), which is somehow expected given that for lower numbers of threads the risks of rollback thrashing within the optimistic HTM-based processing scheme are lower. On the other hand, throttling does not pay-off for slightly larger event granularity values. In fact, the configuration with 12 microseconds event granularity shows about 10% better speedup when throttling is excluded. This is due to the fact that with larger grain events, throttling (that induces a delay in the safety check of speculatively processed events, which tends to be proportional to the event granularity according to the hill climbing dynamically

selected factor) does not pay-off. On the other hand, the same hill climbing scheme tends to shrink the value of the factor n to zero, hence limiting (as said to 10%) the performance degradation by throttling. Overall, the throttling mechanism look to play a core role in the HTM-based speculative engine especially for extremely reduced event granularity values (say event granularity of a few microseconds).

Another set of experiments has been carried out by relying on a multi-robot (multi-agent) exploration and mapping simulation model, shown in Section 5.2. In our implementation of this model, each simulation object models a squared subregion of an overall region to be explored. We considered 2025 subregions, that are explored in parallel by 16 robots, modeling the scenario of a relatively reduced number of high-qualified agents in charge of the exploration. Each robot has a mean residence time within a subregion of 5 minutes, and for physical constraints it cannot pass though a subregion in less than 30 seconds, a value that determines the lookahead of the model (since a new arrival event in any subregion cannot occur before 30 seconds have elapsed since the arrival in the currently visited subregion). This simulation is aimed at determining the coverage time, depending on the choices that are performed while determining what new subregion to enter. This model is still very fine grain (with 6 microseconds of average event granularity), given that mobility events only entail determining what direction to choose. The results for this application are presented in Figure 5.4. By the data we can see how the HTM-based solution is able to deliver maximum speedup of the order of (slightly less than) 3.5, just when using 4 threads. Also, it provides super linear speedup when employing 3 threads.

Further, the traditional style PDES engine did not provide any reasonable speedup also for this case study, rather a slow-down. This additionally

supports the relevance of our HTM-based proposal.

Finally, in Figure 5.4 we also report a speedup curve achieved while running the HTM-based engine using (at the engine level) a lookahead for event safety detection that has been set to 50% of the actual lookahead of the simulation model. By the data we see that this configuration still provides good speedup, which for the case of 4 threads is slightly less than the 80% of the value observed when employing the real lookahead of the application. This is an additional indication of the usefulness of HTM-based speculation, especially for larger number of threads (as we have already noted for the PHOLD benchmark). Finally, these data show how the HTM-based approach can provide resilience to performance failures in scenarios where the lookahead managed at the level of the simulation engine is an under-estimation of the real one. This might help setting up the engine in scenarios where the determination of the precise lookahead of the application can be a time consuming job (e.g. when not relying on automatic approaches to lookahead extraction [83]).

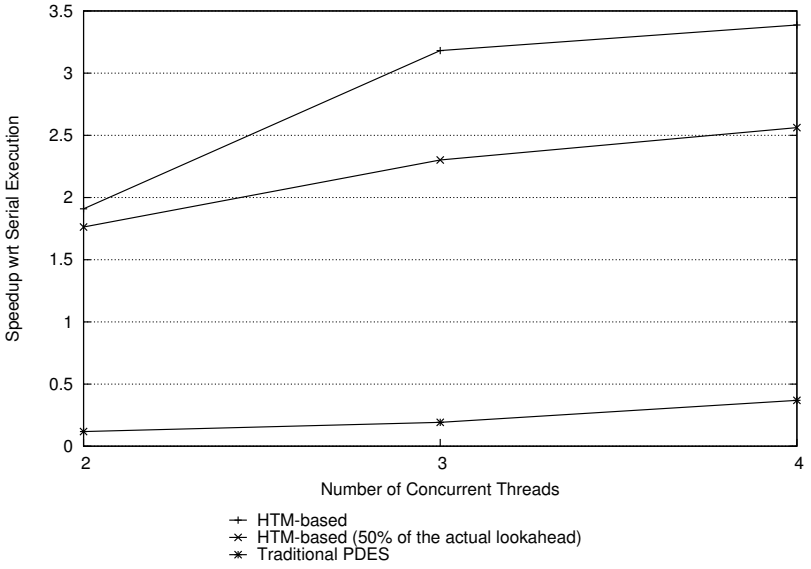

Fig. 5.4: Speedup values for the multi-robot model

Chapter 6

Conclusions

In this thesis we explored the idea of relying on Hardware-Transactional-Memory (HTM) support provided by current off-the-shelf processors in order to improve the execution speed of very fine grain parallel discrete event simulation applications. This is done by exploiting hardware supported in-memory transactions to implement the speculative execution of simulation events. If they are not ensured to be causally consistent at the time of committing the transaction, the rollback operation can be executed at reduced cost by simply squashing the transactional hardware cache. A set of optimizations are included in this proposal, among which an explicit throttling mechanism aimed at improving the chance for a transaction (namely a speculatively executed event) to be positively committable upon the corresponding causal consistency check occurring in the final execution phase of the transaction. Experimental results show that this approach pays off in reducing the relative overhead of the classic software based recoverability support (e.g. software implemented checkpointing) for speculative parallel discrete event simulation. As a result, our scheme allowed to achieve speedup in the parallel execution of discrete event models with event granularity of the order

of a few microseconds, a configuration typically non-effectively addressable via classical speculative parallel discrete event simulation engines relying on software based recoverability.

Chapter 7

Bibliography

[1] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005.

[2] R. M. Fujimoto, "Parallel Discrete Event Simulation," in *Communications of the ACM*, vol. 33 of *WSC*, pp. 19–28, ACM Press, 1989.

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

[4] P. D. Barnes Jr, C. D. Carothers, D. R. Jefferson, and J. M. LaPre, "Warp speed: Executing time warp on 1,966,080 cores," in *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, pp. 327–336, ACM, 2013.

[5] D. E. Martin, T. J. McBrayer, and P. A. Wilsey, "WARPED: A Time Warp Simulation Kernel for Analysis and Application Development," in *HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, p. 383, IEEE Computer Society, 1996.

[6] C. D. Carothers, D. W. Bauer, and S. Pearce, "ROSS: a High Performance Modular Time Warp System," in *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pp. 53–60, IEEE Computer Society, 2000.

[7] A. Pellegrini, R. Vitali, and F. Quaglia, "The ROme OpTimistic Simulator: Core Internals and Programming Model," in *Proceedings of the 4th ICST Conference of Simulation Tools and Techniques (SIMUTools)*, SIMUTools, ICST, 2011.

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

[9] A. Pellegrini and F. Quaglia, "Transparent Multi-Core Speculative Parallelization of DES Models with Event and Cross-State Dependencies," in *Proceedings of the 2014 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, pp. 105–116, ACM, 2014.

[10] R. Rönngren, M. Liljenstam, R. Ayani, and J. Montagnat, "Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation," in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 70–77, IEEE Computer Society, 1996.

[11] A. Pellegrini, R. Vitali, and F. Quaglia, "Autonomic state management for optimistic simulation platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, pp. 1560–1569, June 2015.

[12] R. M. Fujimoto, J. J. Tsai, and G. Gopalakrishnan, "Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp," *IEEE Transactions on Computers*, vol. 41, no. 1, pp. 68–82, 1992.

[13] F. Quaglia and A. Santoro, "Non-Blocking Checkpointing for Optimistic Parallel Simulation: Description and an Implementation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 6, pp. 593–610, 2003.

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

[15] J. M. LaPre, E. J. Gonsiorowski, and C. D. Carothers, "LORAIN: A Step Closer to the PDES 'Holy Grail'," in *Proceedings of the 2nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation*, PADS, pp. 3–14, ACM Press, 2014.

[16] M. Chetlur, N. B. Abu-Ghazaleh, R. Radhakrishnan, and P. A. Wilsey, "Optimizing Communication in Time-warp Simulators," in *Proceedings of the Twelfth Workshop on Parallel and Distributed Simulation*, PADS '98, (Washington, DC, USA), pp. 64–71, IEEE Computer Society, 1998.

[17] A. Gafni, "Space Management and Cancellation Mechanisms for Time Warp," *Tech. Rep. TR-85-341, University of Southern California, Los Angeles (Ca,USA)*, 1985.

[18] R. M. Fujimoto, "Performance of Time Warp Under Synthetic Workloads," in *Proceedings of the Multiconf. on Distributed Simulation*, pp. 23–28, Society for Computer Simulation, 1990.

[19] F. Quaglia and R. Baldoni, "Exploiting Intra-Object Dependencies in Parallel Simulation," *Inf. Process. Lett.*, vol. 70, no. 3, pp. 119–125, 1999.

[20] S. Robinson, *Simulation: The Practice of Model Development and Use*. John Wiley & Sons, 2004.

[21] A. Pellegrini, *Techniques for Transparent Parallelization of Discrete Event Simulation Models*. PhD thesis, Sapienza, University of Rome, 2014.

[22] B. P. Zeigler, *Multifacetted modelling and discrete event simulation*. Academic Press Professional, Inc., 1984.

[23] B. P. Zeigler, "Hierarchical, modular discrete-event modelling in an object-oriented environment," *Simulation*, vol. 49, no. 5, pp. 219–230, 1987.

[24] B. P. Zeigler and T. I. Oren, "Theory of modelling and simulation," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 1, no. 9, p. 69, 1976.

[25] B. P. Zeigler, "On the feedback complexity of automata.," tech. rep., DTIC Document, 1969.

[26] R. G. Sargent, "Verification and validation of simulation models," in *Proceedings of the 37th conference on Winter simulation*, pp. 130–143, winter simulation conference, 2005.

[27] Y. Labiche and G. Wainer, "Towards the verification and validation of devs models," in *Proceedings of the 1st Open International Confer-*

ence on Modeling & Simulation, Clermont-Ferrand, France, pp. 295–305, 2005.

[28] M. H. Hwang and B. P. Zeigler, "Reachability graph of finite and deterministic devs networks," *IEEE Transactions on Automation Science and Engineering*, vol. 6, no. 3, p. 468, 2009.

[29] M. H. Hwang, S. K. Cho, B. P. Zeigler, and F. Lin, "Processing time bounds of schedule-preserving devs," *Techinical Report-2007*, vol. 1, 2007.

[30] M. H. Hwang, "Qualitative verification of finite and real-time devs networks," in *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*, p. 43, Society for Computer Simulation International, 2012.

[31] M. H. Hwang, "Tutorial: verification of real-time system based on schedule-preserved devs," in *Proceedings of 2005 DEVS Symposium*, pp. 2–8, 2005.

[32] G. A. Wainer, *Discrete-event modeling and simulation: a practitioner's approach*. CRC Press, 2009.

[33] J. O. Henriksen, R. M. O'Keefe, C. D. Pegden, R. G. Sargent, B. W. Unger, and D. W. Jones, "Implementations of time (panel)," in *Proceedings of the 18th conference on Winter simulation*, pp. 409–416, ACM, 1986.

[34] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990.

[35] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 652–686, 1985.

[36] R. Brown, "Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem," *Communications of the ACM*, vol. 31, no. 10, pp. 1220–1227, 1988.

[37] T. Dickman, S. Gupta, and P. A. Wilsey, "Event pool structures for pdes on many-core beowulf clusters," in *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, pp. 103–114, ACM, 2013.

[38] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE–S5, no. 5, pp. 440–452, 1979.

[39] R. Simmonds, R. Bradford, and B. Unger, "Applying parallel discrete event simulation to network emulation," in *Proceedings of the fourteenth workshop on Parallel and distributed simulation*, pp. 15–22, IEEE Computer Society, 2000.

[40] M. P. I. FORUM, "Message Passing Interface Forum." \url{http://www.mpi-forum.org/}, 1994.

[41] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. W. LIFT, "A low-overhead practical information flow tracking system for detecting general security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 135–148, 2006.

[42] L.-l. Chen, Y.-s. Lu, Y.-P. Yao, S.-l. Peng, and L.-d. Wu, "A Well-Balanced Time Warp System on Multi-Core Environments," in *Pro-*

ceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation, PADS, pp. 1–9, IEEE Computer Society, 2011.

[43] D. Jagtap, N. B. Abu-Ghazaleh, and D. Ponomarev, "Optimization of Parallel Discrete Event Simulator for Multi-core Systems," in *Proceedings of the International Parallel and Distributed Processing Symposium*, pp. 520–531, IEEE Computer Society, 2012.

[44] R. Vitali, A. Pellegrini, and F. Quaglia, "Assessing Load Sharing within Optimistic Simulation Platforms (invited paper)," in *Proceedings of the 2012 Winter Simulation Conference*, WSC, Society for Computer Simulation, 2012.

[45] R. Vitali, A. Pellegrini, and F. Quaglia, "A Load Sharing Architecture for Optimistic Simulations on Multi-Core Machines," in *Proceedings of the 19th International Conference on High Performance Computing*, HiPC, pp. 1–10, IEEE Computer Society, 2012.

[46] R. Vitali, A. Pellegrini, and F. Quaglia, "Load sharing for optimistic parallel simulations on multi core machines," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, p. 2, 2012.

[47] P. F. Reynolds Jr., "A Spectrum of Options for Parallel Simulation," in *Proceedings of 1988 Winter Simulation Conference*, pp. 325–332, Society for Computer Simulation, 1988.

[48] R. E. Bryant, "Simulation of packet communication architecture computer systems," 1977.

[49] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, vol. 24, no. 4, pp. 198–206, 1981.

[50] H. M. Soliman and A. S. Elmaghraby, "An Analytical Model for Hybrid Checkpointing in Time Warp Distributed Simulation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 947–951, 1998.

[51] S. Srinivasan and P. F. Reynolds Jr., "Elastic Time," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, pp. 103–139, Apr. 1998.

[52] R. M. Fujimoto and D. M. Nicol, "State of the art in parallel simulation," in *WSC '92: Proceedings of the 24th conference on Winter simulation*, pp. 246–254, ACM Press, 1992.

[53] V. Jha and R. Bagrodia, "Simultaneous Events and Lookahead in Simulation Protocols," *ACM Transactions on Modeling and Computer Simulation*, vol. 10, no. 3, pp. 241–267, 2000.

[54] D. M. Nicol, *Parallel discrete-event simulation of FCFS stochastic queueing networks*, vol. 23. ACM, 1988.

[55] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th annual symposium on Computer Architecture*, pp. 135–148, IEEE Computer Society Press, 1981.

[56] F. Chang and G. A. Gibson, "Automatic i/o hint generation through speculative execution," in *OSDI*, vol. 99, pp. 1–14, 1999.

[57] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[58] M. P. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 124–149, 1991.

[59] J. I. Leivent and R. J. Watro, "Mathematical foundations for time warp systems," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, no. 5, pp. 771–794, 1993.

[60] D. R. Jefferson, "Virtual time II: storage management in conservative and optimistic systems," in *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, (New York, NY, USA), pp. 75–89, ACM, 1990.

[61] D. W. Bauer, G. Yaun, C. D. Carothers, M. Yuksel, and S. Kalyanaraman, "Seven-O'Clock: A New Distributed GVT Algorithm Using Network Atomic Operations," in *Proceedings of the 19th Workshop on Parallel and Distributed Simulation*, pp. 39–48, IEEE Computer Society, 2005.

[62] D. West and K. Panesar, "Automatic Incremental State Saving," in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 78–85, IEEE Computer Society, 1996.

[63] Y.-B. Lin and E. D. Lazowska, "Processor Scheduling for {T}ime {W}arp Parallel Simulation," in *Proceedings of the 23rd {SCS} Multiconference on Advances in Parallel and Distributed Simulation*, pp. 11–14, IEEE Computer Society, 1991.

[64] Y.-B. Lin and E. D. Lazowska, *Reducing the saving overhead for Time Warp parallel simulation*. University of Washington Department of Computer Science and Engineering, 1990.

[65] H. Bauer and C. Sporrer, "Reducing rollback overhead in time-warp based distributed simulation with optimized incremental state saving,"

in *Simulation Symposium, 1993. Proceedings., 26th Annual*, pp. 12–20, IEEE, 1993.

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

[67] O. Dalle, "Using tm for high-performance discrete-event simulation on multi-core architectures," in *Using TM for high-performance discrete-event simulation on multi-core architectures*, pp. 1–45, IEEE, 2013.

[68] E. W. Lynch and G. F. Riley, "Hardware supported time synchronization in multi-core architectures," in *Principles of Advanced and Distributed Simulation, 2009. PADS'09. ACM/IEEE/SCS 23rd Workshop on*, pp. 88–94, IEEE, 2009.

[69] S. Srinivasan, M. J. Lyell, P. F. Reynolds Jr, and J. Wehrwein, "Implementation of reductions in support of pdes on a network of workstations," *ACM SIGSIM Simulation Digest*, vol. 28, no. 1, pp. 116–123, 1998.

[70] J. A. Hay, *Experiments with Hardware-based Transactional Memory in Parallel Simulation*. PhD thesis, University of Cincinnati, 2014.

[71] B. P. Swenson and G. F. Riley, "A New Approach to Zero-Copy Message Passing with Reversible Memory Allocation in Multi-core Architectures.," in *PADS*, pp. 44–52, 2012.

[72] S. Bellenot, "Performance of a riskfree Time Warp operating system," in *Proceedings of the 7th Workshop on Parallel and distributed simulation*, pp. 155–158, ACM Press, 1993.

[73] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to os interference using kernel-level noise injection," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, p. 19, IEEE Press, 2008.

[74] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller, "Scheduling support for transactional memory contention management," in *ACM Sigplan Notices*, vol. 45, pp. 79–90, ACM, 2010.

[75] S. Seelam, L. Fong, A. Tantawi, J. Lewars, J. Divirgilio, and K. Gildea, "Extreme scale computing: Modeling the impact of system noise in multi-core clustered systems," *Journal of Parallel and Distributed Computing*, vol. 73, no. 7, pp. 898–910, 2013.

[76] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, pp. 223–234, ACM, 2011.

[77] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," *Communications of the ACM*, vol. 17, no. 8, pp. 453–455, 1974.

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

[79] R. M. Fujimoto, "Exploiting Temporal Uncertainty in Parallel and Distributed Simulation," in *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp. 46–53, IEEE Computer Society, 1999.

[80] A. C. Palaniswamy and P. A. Wilsey, "Adaptive checkpoint intervals in an optimistically synchronised parallel digital system simulator," in *Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration*, pp. 353–362, North-Holland Publishing Co., 1993.

[81] R. Vitali, A. Pellegrini, and F. Quaglia, "Benchmarking memory management capabilities within ROOT-Sim," in *Proceedings - IEEE International Symposium on Distributed Simulation and Real-Time Applications, DS-RT*, pp. 33–40, 2009.

[82] S. Koenig and Y. Liu, "Terrain coverage with ant robots: a simulation study," in *Proceedings of the fifth international conference on Autonomous agents*, AGENTS, pp. 600–607, ACM, 2001.

[83] E. Deelman, R. Bargodia, R. Sakellariou, and V. Adve, "Improving lookahead in parallel discrete event simulations of large-scale applications using compiler analysis," in *Parallel and Distributed Simulation, 2001. Proceedings. 15th Workship on*, pp. 5–13, IEEE, 2001.