Cross-State Events: a New Approach to Parallel Discrete Event Simulation and its Speculative Runtime Support

Alessandro Pellegrini[a], Francesco Quaglia[b,*]

[a]*DIAG, Sapienza, University of Rome, Via Ariosto 25, 00185 Roma, Italy*
[b]*DICII, University of Rome "Tor Vergata", Viale del Politecnico 1, 00133 Roma, Italy*

Abstract

We present a new approach to Parallel Discrete Event Simulation (PDES), where we enable the execution of so-called *cross-state events*. During their processing, the state of multiple concurrent simulation objects can be accessed in read/write mode, as opposed to classical partitioned accesses. This is done with no pre-declaration of this type of access by the programmer, hence also coping with non-determinism. In our proposal, cross-state events are supported by a speculative runtime environment fully transparently to the application code. This is done through an ad-hoc memory management architecture and an extension of the classical Time Warp synchronization protocol. This extension, named Event and Cross-State (ECS) synchronization, ensures causally-consistent speculative parallel execution of discrete event applications by allowing all events to observe the snapshot of the model execution trajectory that would have been observed in a timestamp-ordered execution of the same model. An experimental assessment of our proposal shows how it can significantly reduce the application development complexity, while also providing advantages in terms of performance.

Keywords: Discrete Event Simulation, Parallelization Techniques, Synchronization Transparency, Multicore Computing

1. Introduction

Traditionally, Parallel Discrete Event Simulation (PDES) has been based on the explicit partitioning of the entire simulation model into distinct simulation objects (or Logical Processes—LPs) [1] to be dispatched concurrently. Simulation objects' states are disjoint, and memory accesses during event processing are restricted to the state of the simulation object which is executing the event. Interactions across concurrent objects are only supported

*Corresponding Author

Email addresses: pellegrini@diag.uniroma1.it (Alessandro Pellegrini), francesco.quaglia@uniroma2.it (Francesco Quaglia)

by exchanging timestamped simulation events via messages. Proper synchronization mechanisms are used in order to ensure timestamp-ordered execution of events at each individual object, which is a sufficient (although not necessary [2]) condition for causal consistency.

Such classic approach fits traditional computing environments based on (large-scale) clusters of machines that embed loosely coupled CPUs. Indeed, partitioning the simulation model was the only feasible approach to exploit the computing power offered by this type of (distributed-memory) platforms, in order to speedup the execution of demanding applications.

More recent architectural trends slide towards multi/many-core technology. Here, the same physical memory is directly shared across a (large) set of tightly-coupled cores. This set can be large enough to provide an individual machine with the computing power needed to cope with demanding discrete event simulation models. This paradigm shift has driven the reshuffle of last-generation PDES platforms, in order to meet the peculiarities and potential of highly parallel shared-memory machines. Particularly, this has been done by relying on multi-threading, where all the worker threads operating within the PDES system share a common address space and can take care of executing simulation events scheduled to any object. However, literature proposals along this direction have been mostly focused on how to improve the runtime behavior of the PDES system via the optimization of *platform-level* tasks. Some of these proposals target the management of event exchange across simulation objects (e.g. via shared-memory suited approaches [3, 4]), or the (dynamic) binding of the simulation objects to the different worker threads depending on their actual computation demand (e.g. via load-sharing paradigms [5]).

Yet, none of these proposals had reflections on how PDES applications are actually coded. In fact, in the supported programming models each object still has its own disjoint state, and the execution of an individual event only allows to access the state of the destination object in data separation with respect to the other objects. In other words, no possibility is provided to access the state of multiple simulation objects, while atomically processing an individual event.

In this article, we introduce a new programming model to code PDES applications, along with the runtime support to make consistent their concurrent execution on parallel architectures. In this model, simulation objects can access, while processing any simulation event, the state of an arbitrary set of other simulation objects. In other words, our proposal augments the traditional PDES paradigm with so called *cross-state events*, which take place as atomic actions (potentially) touching in read and/or write mode any memory location currently associated with the memory image of any simulation object. The runtime support to handle cross-state events allows an effective exploitation of parallelism, since it supports the concurrent execution of both traditional and cross-state events on a multi-threaded PDES environment.

The motivations behind our proposal already stand out in the literature. Several results exactly point to the importance of allowing shared access to the same data in PDES systems while concurrently processing events at different simulation objects (see, e.g., [6, 7]). In this sense, cross-state events avoid the burden to pass (large) data structures produced during the execution of some event at simulation object A as the payload of a newly-scheduled event

destined to object B. Indeed, the data can be directly accessed (e.g. via pointers) from the state of object A while processing the scheduled event at the destination object B. This simplifies the job of the application programmer, avoiding the need to manually implement complex marshalling operation of the (possibly scattered) data to be sent as payload of the scheduled event.

Even more important, cross-state events allow to dynamically "cluster" different simulation objects at the logical execution point of a single event, so that application coding can be extremely simplified. This is because any activity (e.g. simulation-state transition) logically involving more that one object can be implemented in a single event handler, atomically accessing the states of all the objects involved in the cluster, rather than being partitioned into multiple activities that are coordinated via explicit event scheduling across the objects—this is what occurs in classic PDES, based on full state disjointness.

At the same time, guaranteeing a correct (i.e. causally-consistent) execution when dealing with cross-state events requires proper application-transparent synchronization mechanisms to be put in place, which we describe in this article. Overall, the contributions in this article can be summarized as follows:

1. We describe the methodology behind the fully-transparent runtime detection of cross-state events, involving any number of simulation objects. This is done taking as a reference x86-64 architectures running Linux, although the concepts can be easily adapted to different architectures and operating systems—all the architectural facilities which are exploited are available on most systems, nowadays. As it will be shown, the runtime identification of cross-state events is fundamental, when application-transparency is pursued.
2. We discuss a synchronization protocol for speculative (optimistic) PDES systems, which we name ECS (Events and Cross-State) synchronization, which takes into account both traditional and cross-state events. This protocol, which can be seen as an extension of classical Time Warp synchronization [8], allows the parallel run to behave in a way which is equivalent to one where all simulation events are processed in non-decreasing timestamp order at any individual simulation object they touch, while being allowed to access any valid memory location belonging to the state of the whole simulation model—namely, any memory location logically belonging to the state of some object. At the same time, our protocol allows to run simulation events destined to different simulation objects concurrently, still transparently to the programmer.

Both the cross-state event detection methodology and the synchronization protocol have been implemented[1] within ROOT-Sim (the ROme OpTimistic Simulator), an open source simulation library supporting the development and execution of PDES models relying on ANSI/ISO-C programming.

The remainder of this article is organized as follows. In Section 2 we briefly discuss the fundamentals of Time Warp synchronization for PDES. The management of cross-state

[1]The source code is available at `https://github.com/HPDCS/ROOT-Sim`.

events and the ECS synchronization scheme are presented in Section 3. The results of an experimental assessment are provided in Section 4. In Section 5 we discuss related literature results.

2. Recap of Time Warp Synchronization

In this section, we briefly present the foundations of Time Warp synchronization. The interested reader is invited to find a more comprehensive discussion in [1, 8].

In PDES systems, each simulation object is associated with its own view of simulation time, known as *Local Virtual Time* (LVT), which is used to locally track the advancement of the computation along the logical-time axis. The execution of an event can mark an update to its state, and moves the LVT to the timestamp of the processed event. It is the responsibility of the underlying runtime environment to track changes of the LVT of some object when dispatching the event to be processed [1]. While processing some event, new timestamped events destined to any concurrent object can be generated and injected into the system. At the same time, state transitions caused by event processing at the different objects can occur concurrently.

Classical PDES correctness rules are based on enforcing non-decreasing timestamp-ordered state transitions at the different simulation objects [1]. Under this enforcement, each simulation object in the system has a coherent view of the flow of logical time.

In the speculative (optimistic) approach to synchronization, also known as Time Warp [8], events are stored by the runtime environment into per-simulation-object event lists, each of which is logically partitioned into a *future-event list* and a *past-event list*. The future-event list stores events not yet processed, while the past-event list records already-processed events. Each simulation object is eligible for dispatching along some thread running within the PDES platform unless its future-event list is empty. Once dispatched, the object is allowed to process the minimum-timestamp event kept by its future-event list. Such an event is then moved to the past-event list.

In Time Warp, a simulation object is scheduled for execution without any safety verification of causal consistency of its next-to-be-processed event. Hence, timestamp-order violations might arise, since a simulation object may receive an event (produced by another object) with a timestamp lower than its LVT. If a timestamp-order violation is detected, all the events that were executed out of timestamp order are rolled back by the runtime environment—they are moved back from the past-event list to the future-event list. Also, the LVT of the object is pushed back to the timestamp of the last event executed in correct order, and the object's state is restored to its value prior to the timestamp order violation, which is achieved by either relying on traditional checkpointing methods (see, e.g, [9–11]) or by the means of reverse computing approaches (see, e.g., [12, 13]), where reverse versions of the event processing routines are executed when rolling back processed events.

For the classical scenario where simulation-object states are disjoint, restoring the system to a correct state entails restoring individual object states independently, with no risk of domino effect. In particular, if dependencies occurred due to the scheduling of some event between a rolling back object O_a and another one O_b, these dependencies are undone via

so called *anti-events*[2]. More in detail, an anti-event is generated for each event injected in the system by O_a during the portion of the computation to be rolled back, and is used to retract the originally injected event. Upon the delivery of an anti-event associated with an already executed event by O_b, the recipient also rolls back. Instead, if the event has not yet been executed, the anti-event has the only effect to "annihilate" the originally injected event. This operation typically occurs within the runtime environment. After rolling back, any simulation object resumes the execution of the events from its future-event list.

A concept that is relevant to optimistic synchronization is Global Virtual Time (GVT), which is defined as the smallest timestamp among those of (i) unexecuted events already inserted into the simulation objects' event lists, (ii) events being executed, (iii) events/anti-events in transit from source to destination. Since no simulation object can ever rollback to a logical time preceding the GVT [8], the GVT value indicates the commitment horizon of the speculative computation. It is used both to execute actions that cannot be subject to rollback, such as displaying of intermediate results [14, 15], and for recovering memory. Specifically, event buffers with timestamps lower than the GVT value will never be reused after a rollback, therefore the runtime environment can discard them from the past-event lists of the simulation objects. The same happens to obsolete state information, if any, maintained to support state recoverability. The action of recovering memory after GVT calculation is typically referred to as *fossil collection*.

As hinted, in our approach we target optimistic PDES platforms relying on the multi-threading paradigm, which have been shown to provide a set of benefits and to support optimized resource usage policies (see, e.g., [4, 16–18]) when compared to the traditional counterpart where parallelization is achieved by running a set of single-threaded processes within the simulation platform. Hence, we target the scenario where multiple threads can take care of dispatching whichever simulation object for execution which takes place by simply calling an event handler, with proper input parameters, along that thread. It is still possible to rely on temporary binding schemes between objects and worker threads in order to cope with, e.g., locality and other performance-related aspects.

3. Cross-State Events

3.1. Outline

The programming model enabled by cross-state events allows to bypass state disjointness in event processing. This enriches the traditional PDES programming model significantly. We provide such a support fully transparently to the programmer, who is therefore allowed to design simulation models able to access memory locations within any simulation objects' state in read/write mode. The programmer is also provided with the illusion that memory accesses are performed as if the events were dispatched for processing in non-decreasing timestamp order. In other words, the event-handling routine can be coded under the assumption that any memory access it performs is related to the current simulation time of

[2]An anti-event is a "negative copy" of the corresponding event, or of its digest.

the event dispatched for processing. This corresponds to the LVT of the simulation object targeted by the event. On the other hand, we still enable parallelism in the execution in combination with speculation.

To make the programming model clearer, we show in the code snippet below a C example of an event handler in a PDES model implemented by relying on cross-state events. The event handler takes control as a callback function invoked by the underlying PDES runtime library. As typical of PDES-style coding rules, this function has a set of input parameters: the identifier of the object (`obj`) for which event processing is taking place and the state base pointer (`state`) which is the address of the data structure starting from which the object is allowed to access any other dynamically-allocated buffer belonging to its state via pointers. It also takes the timestmap of the currently dispatched event, and the event payload.

```
1  void ProcessEvent(object_id obj, state_t *state, time_t event_timestamp, event_payload_t event) {
2
3          char *p, q;
4          ...
5          p = event.reference;
6
7          q = *p; // reading the content of memory location pointed by the 'reference' field in the event payload
8          ...
9          *p = q ; // writing the content of memory location pointed by the 'reference' field in the event payload
10         ...
11 }
```

This can result in a cross-state event because we find statements where in-place pointer-based accesses to memory occur both in read (line 7) and write mode (line 9). In the example, the target memory locations for the accesses are taken from the payload of an event that has been scheduled for the simulation object identified by `obj`. These memory locations might correspond to memory buffers included in the state of the object that generated the event, which can be different from the one which runs it, identified by `obj` in the code snippet. More generally, a cross-state event is a simulation event scheduled for some simulation object A, for which the corresponding processing steps lead to accessing in read/write mode the state of one or multiple simulation object(s) distinct from A.

This is an innovative programming model not supported by any literature proposal. Indeed, reference PDES systems (e.g. [19, 20]) allow the programmer to rely only on the base pointer to the state of the simulation object that is the target of the event (namely `state`) to implement memory read/write operations. Also, the above example shows how in our solution we do not require any annotation in the code of the event handler. In other words, the programmer does not need to discriminate whether a memory access will refer to the local state of the simulation object targeted by the event, or to some remote object. We do not impose/require any a-priori indication of whether some event scheduled by the PDES application is a cross-state one or not. Hence we rely (and enable) pure runtime tracking of the occurrence of such type of events while the PDES run is in progress. Overall, the support for the correct handling of the memory accesses is fully transparent in our proposal, thus masking to the application developer any complication related to the possibility for two different objects targeted by a cross-state event to be concurrently scheduled for execution along two different threads.

Our solution is therefore significantly different from shared-data management approaches

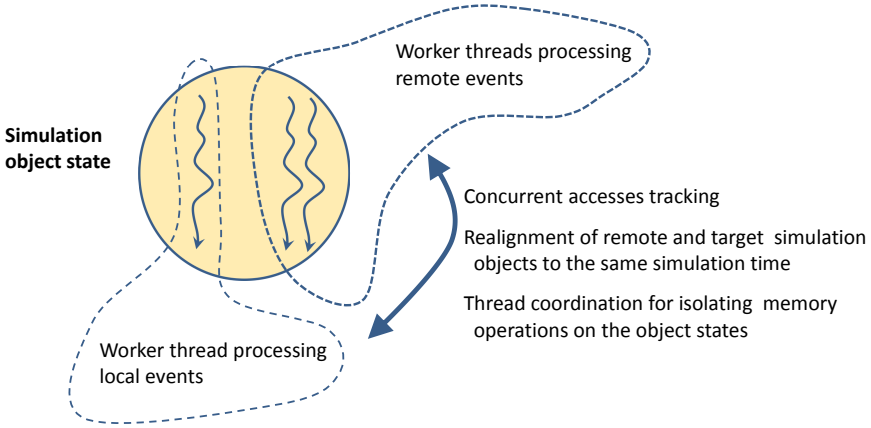

Figure 1: A scheme of state accesses to a simulation object with cross-state events.

like, e.g., Transactional Memory (TM), where code portions that might just potentially access the same data concurrently along different threads must be explicitly annotated. This is for example what happens when relying on the TM support offered by the `gcc` compiler.

Even more important, TM-style synchronization mechanisms to handle concurrent shared-data accesses are devised to achieve data consistency via isolation. Rather, a cross-state event does not only need to be managed via isolated accesses involving multiple objects along a same thread. It also needs to access object states that are aligned in simulation time to the timestamp of the cross-state event—namely, the LVT of the object processing the event. This is another reason for common TM support to be unsuited for the purpose of speculative PDES with cross-state events.

In Figure 1, we provide a schematic representation of the capabilities of the approach we propose. As depicted, we enable a single simulation object to advance in simulation time (still according to a speculative scheme), and to be the target of concurrent threads' memory operations. One of these threads is in charge of dispatching events destined to that simulation object—the simulation object is currently bound to that thread. These events are denoted as "local events" in the picture. On the other hand, other threads in charge of dispatching other simulation objects can anyhow process some cross-state events that require to access the state of the former simulation object. These events lead to a kind of remote access by those threads. In our proposal we provide mechanisms allowing:

A. the transparent runtime detection of remote accesses;

B. the realignment of the state of the simulation object targeted by a remote access to the simulation time corresponding to the timestamp of the cross-state event that determines the remote access, and the correct operation of threads in their concurrent activities within the extended Time Warp synchronization involving cross-state events.

Point A is addressed via a PDES-oriented state management architecture that controls

the way memory is delivered to keep the states of the simulation objects, and performs operating system tasks enabling the PDES runtime environment to track those remote accesses caused by the processing of cross-state events. In our solution we deal with object states kept into the heap, although we do not explicitly consider Thread Local Storage (TLS). This is because the state of the simulation objects in modern and general-purpose multi-threaded PDES systems is intrinsically not private to specific threads, since objects can be dynamically re-bound to different threads for load-balancing purposes (see, e.g., [21, 22]). Our state management architecture is presented in Section 3.2. Point B is dealt with via the extension of the Time Warp synchronization protocol, which we name ECS synchronization. This is presented in Section 3.3. We emphasize again that, although our proposal targets Linux systems on the x86-64 architecture, the concepts it is based on are general and valid for other systems and architectures.

An important point to emphasize is that the state management architecture we present only deals with accesses to virtual addresses, and is fully transparent to the actual physical placement of the virtual pages hosting the states of the simulation objects. This implies that literature optimizations for placing virtual pages onto physical frames can be combined with our support for cross-state events. Indeed, our integration of this support within the ROOT-Sim platform coexists with the NUMA (Non Uniform Memory Access) optimizations already present in this runtime library [23]. With respect to this point, we stress again that our state management architecture is not a competitor of literature proposals for optimizing memory accesses in PDES on multi-core machines (like for example the ones presented in [3, 20]). Rather, it is an orthogonal means to enable an application programming model that was not supported by any literature proposal. Overall, our primary objective is to provide the programmer with a flexible way to develop models, while still guaranteeing adequate performance by the parallel run.

3.2. Object States Management and Memory Access Tracking

In our state management architecture, virtual memory is destined for usage by any simulation object (e.g. for allocating buffers belonging to the object state) according to *stocks*. More in detail, when the object requests new memory buffers, the memory management architecture reserves an interval of page-aligned virtual memory addresses (the stock) via the standard `mmap` POSIX API. Memory allocation is supported to the application programmer via the standard `malloc` API, transparently redirected at compile/link time to a proper memory allocator—see Appendix A for details.

To understand how the stock abstraction can be used to track memory accesses related to cross-state events, let us consider the case of the IA32e paging scheme offered by x86-64 architectures—this is the paging scheme of x86-64 processors which is exploited by 64-bit versions of Linux. As shown in Figure 2, any 64-bit logical address has only 48 valid bits, which are used as access keys for a 4-level paging scheme, ultimately supporting pages of 4KB in size. The top level page table is called PML4 (or also PGD—Page General Directory) and keeps 512 entries. All the other page tables, operating at lower levels, also have 512 entries each.

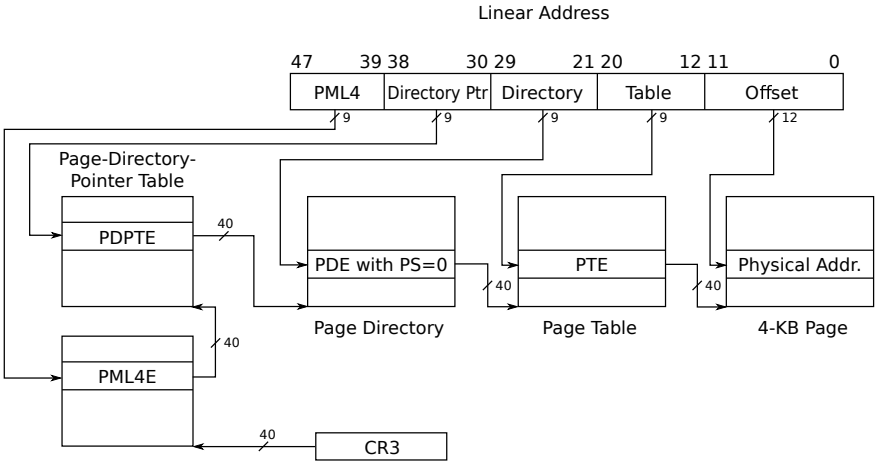

Figure 2: The IA32e paging scheme with 4KB pages in x86-64 processors.

In our design, the stock of virtual memory pages used to allocate memory buffers for the state of a given simulation object corresponds to the set of contiguous virtual pages whose virtual-to-physical memory translation is associated with a single entry of the second-level page table, which is called PDP (Page Directory Pointer), and whose entries are referred to as PDPTE. Note that a single stock corresponds to 512^2 pages, for a total of 1GB of virtual memory. Hence, a single stock allows to manage object states up to 1GB of (dynamic) memory. On the other hand, reserving multiple stocks for a same simulation object will lead to manage object states reaching multiple gigabytes in size.

To interact with this memory management system, our architecture relies on a driver implemented into a Loadable Kernel Module for the Linux kernel. This driver creates a special device file which can be handled via ad-hoc `ioctl` commands supported by our driver. The `SET_VM_RANGE` command allows the driver to register the stocks to be reserved, and their association to the simulation objects (which are identified via classical unique numerical identifiers, as in typical PDES platforms). When this command is issued, the state of driver changes, and a kernel-level map, accessible in constant time, is set up. Here, for each reserved stock (logically related to one entry of a PDP page-table) the identifier of the simulation object destined to use that stock is recorded. In Figure 3 we show an example where a given PDP table has its 0-th entry (hence the corresponding stock of virtual memory pages) reserved for object x, and its 1-st entry reserved for object y.

Thanks to this organization, if simulation object x accesses any virtual address included in the stock reserved for object y, we know that such a memory access (which can be either in read or write mode in our execution model) is occurring outside the boundaries of its local state, and is actually involving the state of another object. Therefore, we are in the presence of a cross-state event.

The main problem to cope with to exploit the stocks as the means to capture whether the generic simulation object x (currently dispatched for execution along any worker tread, noted as WT_i, in the PDES platform) is involved in a cross-state event is related to how to determine that event processing gives rise to a memory reference falling outside the bound-

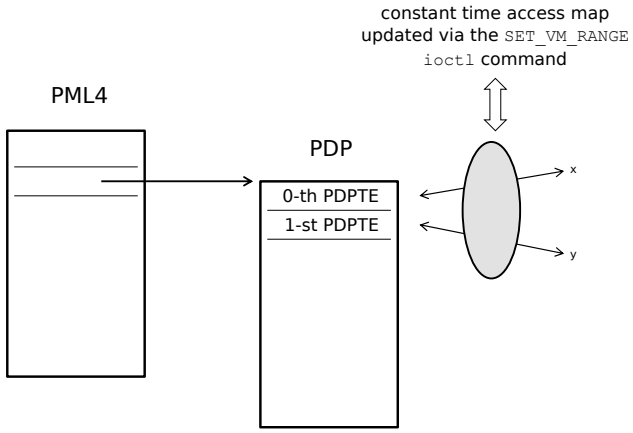

Figure 3: Example of association between stocks of virtual memory pages and simulation objects.

aries of the stocks currently reserved for object x. Classical memory protection mechanisms supported by the operating system (and related segmentation-fault handling schemes) are not suited for our purposes. Indeed, we cannot simply protect a-priori the accesses to stocks that are reserved for simulation objects other than x upon dispatching x along any worker thread WT_i. This is because these simulation objects might be requested to run concurrently with respect to x along other worker threads, which all share the same page table and experience the same protection rules as WT_i. Overall, preventing WT_i to access the stocks not reserved for simulation object x upon dispatching it (e.g. via the `mprotect` POSIX API) would lead to a change of the state of the page table so that *any other thread* would not be allowed to access those stocks. This definitely hampers concurrency, also leading to unneeded memory faults—by threads running objects other than x—in contexts where the object x does not require any access to "remote" stocks while processing the event.

Similarly, addressing the above problem via (user-transparent) code instrumentation would require to instrument memory-write[3] and also memory-read instructions. This would lead to (non-minimal) overhead to be paid even in case no cross-state access, although admissible, will ever occur—this renders the tracking of memory operations pointless on the side of synchronization tasks related to the management of cross-state events.

To cope with this issue, we have devised a memory management architecture where any worker thread WT_i is associated with a sibling PML4 page table, whose entries point to sibling PDP page tables. By default, the entries of the sibling PDP page tables, which are associated with the stocks that have been destined for usage by the simulation objects, are all set to NULL. This means that they do not allow to reach the lower-level page tables, hence not allowing access to any already allocated stock. Therefore, any attempt to access the stocks will lead to a memory fault. On the other hand, when WT_i dispatches simulation object x for event execution, the entries of the PDP sibling tables that correspond to the virtual-memory stocks destined for usage by x are "opened" to correctly allow the retrieval

[3]This is typically done when supporting transparent incremental checkpointing in speculative PDES platforms [24, 25].

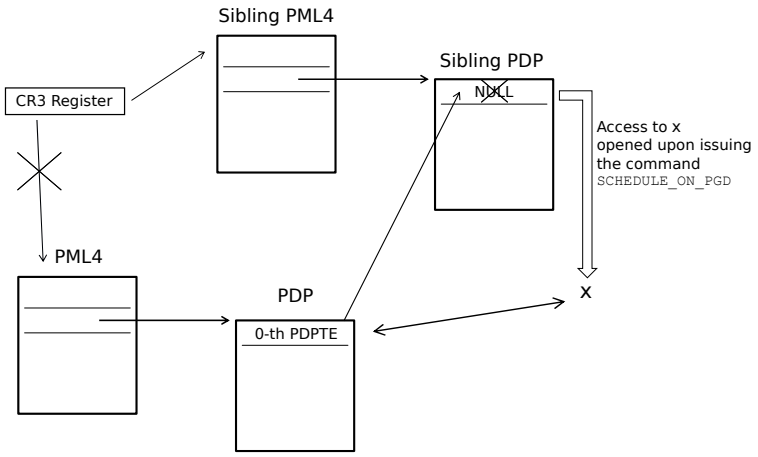

Figure 4: Example scenario where the memory stock associated with simulation object x is opened for access onto a sibling PDP page table.

of the lower-level page tables that contain the actual virtual-to-physical memory mapping (or tell that the pages are not present, e.g., they are swapped-out pages). This is done by copying the corresponding entries of the original PDP tables onto the destination entries of the sibling PDP page tables—see Figure 4 for an example scenario where the stock associated with simulation object x is again related to the 0-th entry of a given PDP page table. In our architecture, this operation can be executed via the `ioctl` `SCHEDULE_ON_PGD` command.

By using this command, the worker thread switches to what we refer to as *simulation-object mode*, where the only accessible stock is the one associated with the dispatched object (x in the example discussion), while the other stocks are not accessible in any way, given that their corresponding entries into the sibling PDP page tables are still set to NULL. As depicted in Figure 4, this operation also updates the `CR3` register (namely, the page table pointer register in `x86-64` processors). This allows to switch to the sibling PML4 for virtual-to-physical address resolution purposes.

Having different sibling PML4 tables, associated with the different concurrent worker threads, allows to concurrently dispatch and execute different simulation objects—this is done by having each worker thread open the access to the stocks associated with the object it is currently dispatching—while still having the possibility to determine whether any of the dispatched objects is keeping its memory references within its own stocks.

Two additional points must be discussed. First, having all the stocks closed for access by the worker thread, except the one(s) related to the dispatched object, leads (as noted before) to memory faults in case of a memory access to stocks other than open one(s), namely in case of the occurrence of a cross-state dependency across concurrent simulation objects. However, these faults cannot be tracked (and handled) via classical segmentation-fault handling given that the "remote" stocks have already been validated via `mmap`, and the Linux kernel would simply lead the fault to reallocate the whole chain of page table entries for mapping the accessed virtual page in memory. This would bring the whole system into an inconsistent state where for the same virtual page multiple chains of page tables (and

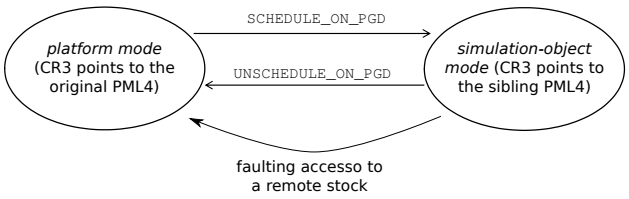

Figure 5: The state diagram for switch operations between original and sibling PML4 page tables.

thus physical frames) are present. This is something not directly handled by the Linux kernel, except if using invasive patches. To prevent this, when loading the external module, we also change the Interrupt Descriptor Table (IDT), directly accessible via the `IDT` register, to make the pointer to the page-fault handler point to a custom handler, rather than the original `do_page_fault` kernel function. If the fault is not related to accesses to remote stocks within the sibling paging scheme, then the original handler is invoked. Otherwise, the custom handler pushes control back to user mode in order to let the PDES platform actuate synchronization policies to handle cross-state events, as we shall discuss later when presenting ECS synchronization.

Upon a memory-fault occurring on sibling PDP entries because of cross-state dependency materialization, the faulting thread is put back into what we call *platform mode*, which implies that it is switched back to the original PML4. This is done to let this thread operate any memory access required to reconcile the execution of the concurrent objects according to ECS synchronization. This aspect will be treated in detail in the next section. On the other hand, if event processing at the currently-dispatched object ends, the worker thread can switch back to platform mode on demand—hence gaining again access to any memory location or data structure supporting the parallel execution—by using the `ioctl` `UNSCHEDULE_ON_PGD` command implemented within the driver. In Figure 5, we show the state diagram where the events causing the switch between simulation-object and platform modes are depicted.

Second, our architecture needs anyway to co-exist with the kernel scheduler, which poses issues on the side of managing the sibling PML4. Particularly, all the threads within a same Linux process share the same memory management information (the so called memory context), including the pointer to the original page table. This pointer is used by the kernel scheduler upon re-dispatching the thread after it has been context-switched off the CPU. Particularly, this pointer is reloaded into the page-table pointer register `CR3` upon the occurrence of a context switch that gives control to the thread. However, if the thread was executing in simulation-object mode, `CR3` would need to be filled with the address of the sibling PML4 (instead of the original page table).

Rather than providing a recompiled version of the kernel directly embedding a custom schedule function for correct management of the `CR3` register (which would reduce the usability of our system), our Linux module adopts a *dynamic patching* approach based on the `kprobe` facility supported by Linux. It allows to inject an execution flow variation such that control goes to a `schedule_hook()` routine in the module right before the kernel's `schedule()` function would execute its finalization part (i.e., stack realignment and return). In

this way, the actual final part of the scheduling process is under the control of our external module. The `schedule_hook()` function simply executes the same return actions originally planned by the kernel's `schedule()` function. However, patching the original scheduler in this way allows the hook to take control when the decision about what thread needs to take control of the CPU-core[4] is already finalized.

The `schedule_hook()` function implements the `CR3` custom manager. This manager checks whether the CPU-scheduled thread is running into simulation-object mode (which can be done by checking per-thread meta-data that were setup via `SCHEDULE_ON_PGD`) and, in the positive case, it loads the sibling PML4 pointer into the `CR3` register (thus persisting in the simulation-object mode when running the thread).

3.3. ECS Synchronization

In this section we describe ECS synchronization, which ensures causal consistency for speculative PDES runs involving cross-state events. ECS synchronization has two goals. First, letting each simulation object process its events in non-decreasing timestamp order, in accordance with the reference correctness criterion for PDES platforms only entailing event dependencies. Second, allowing any cross-state dependency occurring at simulation time t—due to the occurrence of a cross-state event at that time—to let the involved object accessing remote stocks to observe the state snapshot that would have been observed at simulation time t in a timestamp-ordered run.

ECS synchronization can be seen as an extension of Time Warp synchronization based on the following two innovations:

- the introduction of temporary object blocking phases, which may lead to temporary block of the execution of an already dispatched simulation event at a certain object;
- the introduction of so called *rendez-vous events*, which are system-level events not causing updates on the destination object state, rather only driving block and unblock actions for processing activities at the concurrent objects. These are exploited to temporarily disable a simulation object to perform updates to its state along the simulation-time axis, given that its state snapshot is currently being accesses by a cross-state event (namely, it is involved in what we have referred to as a cross-state dependency).

3.3.1. Rendez-vous Events

In our proposal, each simulation object x is associated with a cross-state dependency set which we refer to as CSD_x. It records the identifiers of all the simulation objects towards which x has materialized a cross-state dependency during the processing of a single event. CSD_x is initialized as empty upon dispatching object x for the execution of any new event, and gets possibly updated while processing this same event (if it reveals as a cross-state

[4]It has actually already taken control of the CPU-core, since we are returning from the CPU-scheduling step.

one). ECS synchronization exploits the ad-hoc memory management architecture that we presented in order to detect that simulation object x is accessing a remote memory stock, namely the stock associated with object y, in either read or write mode, while processing its currently dispatched event e_x. The identity of the object towards which the cross-state dependency is being materialized (y in our example discussion) is also known, given that the custom memory-fault handler we designed, which pushes the faulting thread back in platform mode, notifies such an identifier into the thread user-mode stack. The memory fault occurrence gives rise to the following algorithmic steps:

1. The execution of e_x is temporarily suspended by having object x transitioning into a block state;
2. A rendez-vous unique identifier is generated and assigned to the event e_x, which we refer to as $rvid(e_x)$;
3. A special rendez-vous event e_y^{rv} is scheduled for object y, marked with timestamp equal to the timestamp of event e_x, and with its identifier. We formally express this with the notations $ts(e_y^{rv}) = ts(e_x)$ and $rvid(e_y^{rv}) = rvid(e_x)$.

Rendez-vous events are not generated by the application layer, rather they are platform-generated events. Hence they do not have any associated processing rule at the application level. Nevertheless, ECS rendez-vous events are incorporated into the event list of the destination object (kept at the PDES platform level) as if they were traditional events, which is done for synchronization purposes.

Given that we are targeting speculative synchronization, a rendez-vous event may be a straggler event (if its timestamp is lower than the timestamp of some already processed event at the destination simulation object), thus possibly causing a rollback. Overall, rendez-vous events need to be processed at the destination according to timestamp-order (possibly when resuming after a rollback), and the processing actions are platform-level ones given that, as hinted above, no application level processing rule is—and needs to be—specified for rendez-vous events in ECS.

When a simulation object y is dispatched for processing a rendez-vous event e_y^{rv}, ECS performs the following algorithmic steps:

1. Object y is put into a block state;
2. A special rendez-vous acknowledgment event e_x^{rva} is scheduled for object x, marked with no-timestamp but with the same rendez-vous identifier of e_y^{rv}, formally $rvid(e_x^{rva}) = rvid(e_y^{rv})$.

On the other hand, when the rendez-vous acknowledgement event e_x^{rva} is delivered to the destination simulation object x, ECS performs the following steps:

1. It inserts the identifier of the sender object, namely y, into CSD_x.
2. It unblocks the simulation object x so that it can be eventually re-scheduled along some worker thread (which can lead to resuming the execution of the originally-interrupted event e_x).

object_local $identifier\ current\_rendez\_vous$;

upon dispatching event e_x at object x **do**
1. $CSD_x = \emptyset$;
2. $current\_rendez\_vous_x = null$;
3. $rvid(e_x) = null$;

upon faulted access to the memory stock of object y by object x **do**
1. $e_x = get\_current\_event(x)$;
2. **if** $(rvid(e_x) = null)\{$
3. $\quad mark = get\_unique\_id()$;
4. $\quad rvid(e_x) = mark$;
5. $\}$
5. $e_y^{rv} = instantiate\_rendez\_vous\_event()$;
6. $ts(e_y^{rv}) = ts(e_x)$;
7. $rvid(e_y^{rv}) = rvid(e_x)$;
8. $send\ e_y^{rv}\ to\ object\ y$;
9. $current\_rendez\_vous_x = rvid(e_x)$;
10. $block\ execution\ of\ object\ x$;

upon dispatching rendez-vous event e_y^{rv} at object y **do**
1. $source = get\_sender(e_y^{rv})$;
2. $e_x^{rva} = instantiate\_acknowledgment\_event()$;
3. $rvid(e_x^{rva}) = rvid(e_y^{rv})$;
7. $send\ e_x^{rva}\ to\ object\ x$;
8. $current\_rendez\_vous_y = rvid(e_y^{rv})$;
9. $block\ execution\ of\ object\ y$;

upon the receipt of the rendez-vous acknowledgement event e_x^{rva} at object x **do**
1. **if** $(current\_rendez\_vous_x \neq rvid(e_x^{rva}))$ **return**;
2. $source = get\_sender(e_x^{rva})$;
3. $CSD_x = CSD_x \cup source$;
4. $unblock\ object\ x$;

upon ending the execution of event e_x at object x **do**
1. $\forall k \in CSD_x$
2. $\quad e_k^{ub}\ \ send\ _x = instantiate\_unblock\_event()$;
3. $\quad rvid(e_k^{ub}) = rvid(e_x)$;
4. $\quad send\ e_k^{ub}\ to\ object\ k$;

upon the receipt of the unblock event e_k^{ub} at object k **do**
1. $unblock\ object\ k$;

upon the resuming execution of object x after a block phase **do**
1. $\forall k \in CSD_x$
2. $\quad open\ access\ to\ memory\ stock\ associated\ with\ k$;

Figure 6: Pseudo-code for the management of rendez-vous events.

At this point, we know that simulation object y is blocked, thus not being currently allowed to process its events. Hence, the snapshot of its state is available to simulation object x for read/write operations, such as the operation that originally gave rise to the cross-state event's memory fault. However, upon re-dispatching object x (which leads to resuming the processing of e_x), the involved worker thread cannot move into simulation-object mode by only opening the stock(s) associated with x into the sibling page tables. Indeed, we also need to open access to the stock(s) associated with object y. In our architectural support, this can be easily achieved via the `SCHEDULE_ON_PGD` command, since is allows to get a set of identifiers whose stocks need to be opened within the sibling page tables. In particular, when re-dispatching object x, the `SCHEDULE_ON_PGD` command is issued with the set $x \cup CSD_x$ as input, which for our example discussion, contains the identifiers of both objects x and y.

The above algorithmic steps can be iterated in case cross-state dependencies are materialized towards multiple simulation objects while processing the event e_x, which will lead to the scenario where simulation object x can be rescheduled multiple times (while processing e_x) with an incrementally enlarged set of open stocks. Once a remote memory stock (associated with a distinct object) becomes open for access by object x during the processing phase of event e_x, any access to this stock by x will not cause any additional memory fault while processing the same event.

To finalize the processing of e_x, we need to notify that the stocks associated with simulation objects towards which cross-state dependencies have been detected are no longer locked for access by object x. Hence, the owner simulation objects can resume their processing activities. This is done through the following steps, carried out at the end of the execution of e_x at object x:

1. an unblock-event e_k^{ub} is sent towards any object k whose identifier is in CSD_x. These events are again not marked with a timestamp, but with the rendez-vous identifier of the event e_x originating the cross-state dependency;
2. upon the delivery of e_k^{ub}, the recipient simulation object is simply put back as ready for being dispatched (hence exiting the block state).

In Figure 6, we report the pseudocode for the management of rendez-vous events. Two important notes. First, an event which generates a rendez-vous keeps the same *rvid* mark for all the possibly additional rendez-vous it will generate, in order to have a unique logical identifier for the rendez-vous, independently of the actual group of participating objects. Second, when a rendez-vous is triggered by some object x, the *rvid* value is recorded in the per-object variable *current_rendez_vous*, so that if a rendez-vous acknowledgement is received with an *rvid* value that does not match the current one seen by the simulation object, then the processing of the acknowledgment is simply skipped. This is related to the management of rollbacks in the speculative run, which we discuss in the next section.

In Figure 7, we show a timeline illustrating how ECS manages a cross-state event e_x processed by simulation object x. With no loss of generality, the represented scenario refers to the case of two objects involved in the cross-state event, which are handled, in terms of CPU-dispatching, by different threads operating within the PDES system. The red bars indicate the wall-clock-time intervals along which the simulation objects x and y involved

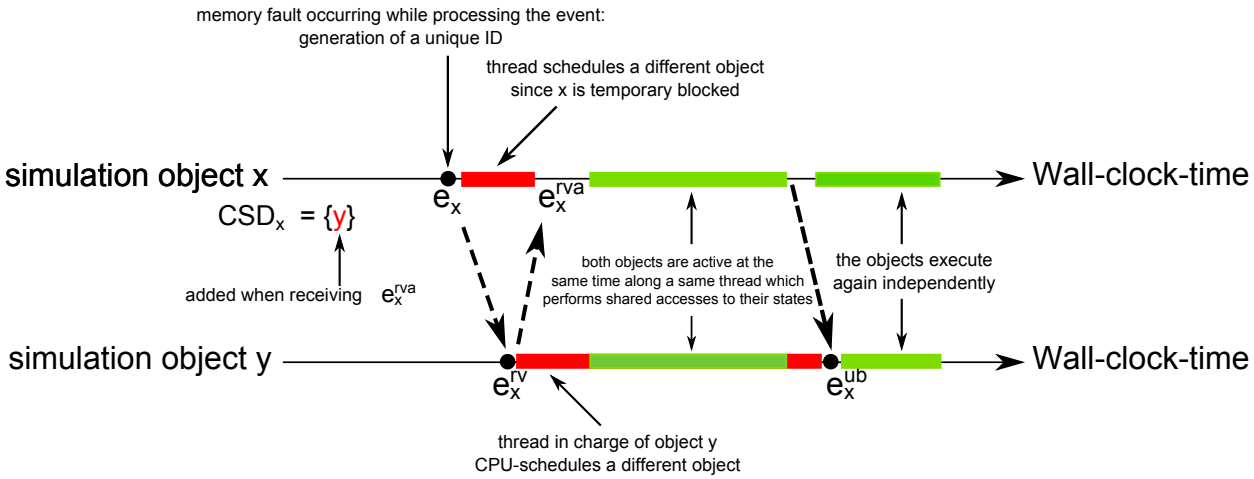

Figure 7: An example time-line of ECS

in the rendez-vous are blocked. Object x is blocked up to the point in time where the state of object y is ready for being consistently accessed by the thread that processes the event e_x. In the meanwhile the thread in charge or processing the events destined to object y can CPU-schedule some object different from y. After the event is processed, object y stays blocked again up to the receipt of the ECS unblock event e_x^{ub}. Afterwards, both the objects can proceed processing their events concurrently along different threads. Clearly, in a scenario where the cross-state event hits a simulation object that is handled by the same thread that is processing the event, all ECS tasks are run along that same thread.

3.3.2. Dealing with Rollbacks

ECS targets speculative processing, where object blocking is never caused by native event dependencies, rather by the need for executing memory read/write operations in multiple stocks as an atomic action (within a rendez-vous). Thus, some care must be taken when handling rollbacks. In particular, when we process an event e_x that gives rise to a rendez-vous event e_y^{rv}, we need to define rules to handle the rollback of either object x or object y at a simulation time $t' < ts(e_x)$—or equivalently $t' < ts(e_y^{rv})$. The peculiarity of this scenario is related to fact that e_x and e_y^{rv} are both causally related to each other. Particularly, if e_x is rolled back, then we need to rollback e_y^{rv} given that object x may have performed updates on the memory stocks destined to keep the state of simulation object y while processing e_x. On the other hand, the processing outcome of e_x is affected by values possibly read by object x from the stocks destined to y at time $ts(e_x)$. If these values change due to a rollback of object y at a simulation time preceding $ts(e_y^{rv})$, the updated values should have been observed while processing e_x by object x.

To handle such a mutual dependency, we devise the following scheme. When the event e_x is rolled back, we simply send a classical anti-event for the rendez-vous event e_y^{rv} that was scheduled while processing e_x. Given that e_y^{rv} was incorporated into the event list of the destination object y, the arrival of the anti-event gives rise to a classical annihilation that

possibly rolls back y to the latest processed event with timestamp less than $ts(e_y^{rv})$.

The only additional aspect to consider is the effect of the rendez-vous anti-event on the destination object y in case it is still locked in the original rendez-vous. This scenario may occur because the source object x may experience a rollback leading to squashing the currently issues rendez-vous—this might be requested for progress of the run. In this case, no unblock event will be notified to y, and the effects of receiving the anti-event will be exactly that of pushing y out of the block state, as if the unblock event for the rendez-vous were received.

On the other hand, if the rollback is originated on object y, and pushes it to a simulation time less than $ts(e_y^{rv})$ (which leads to undoing the execution of e_y^{rv}), the following actions are taken. A special rendez-vous-restart event e_x^{rvr}, marked with the original rendez-vous identifier $rvid(e_x)$ is sent out towards object x. This special event annihilates the processing of the original instance (while not removing it from the input queue), which leads to ultimately undoing e_y^{rv} via an anti-event. When processed after the rollback, e_x will generate a new instance of rendez-vous marked with a new unique identifier, so no mismatch will occur in any annihilation phase for rendez-vous events associated with different incarnations of their generating event. This also avoids cycles in the annihilation process.

Furthermore, all the other types of events used in ECS, such as acknowledgment and unblock events (see Figure 6), are not incorporated into the event lists of the simulation objects. Thus they are inherently ephemeral, and do not require particular care in the rollback scheme. Assuming FIFO communication across the objects (as can be typical of multi-thread PDES platforms [4]), these events can be simply discarded at the recipient side if the rendez-vous associated with their corresponding identifier (e.g. $rvid(e_x)$ in case of the acknowledgement event sent to x upon the rendez-vous) is no longer in place.

Appendix B discusses correctness aspects of the presented ECS rollback scheme, particularly by the side of progress.

4. Experimental Study

4.1. Test-bed Platform

We integrated ECS within the open source ROOT-Sim package [26], particularly the symmetric multi-threaded version presented in [4]. A few relevant modifications to this simulation platform have been made for integration purposes. Most relevantly, we have created stack separation across the different simulation objects, by locating the stack of each object in the initial part of a stock of memory destined for object usage. This is done in a fully application-transparent manner. Further, execution resume in the different stacks, by also providing the correct processor and stack image, has been supported via modified versions of the `setjmp` and `longjmp` POSIX APIs. These ad-hoc APIs take into account the fact that an event can be theoretically descheduled at any memory-access instruction, including points which are not "safe" from a calling convention point of view. Further details on these aspects are discussed in Appendix C. They have also been used as the support to squash the stack when a rollback occurs while the simulation object is in the block state—this eventually leads the object to resume execution in a different context.

The application transparent facilities that have been used in our implementation to save and restore the states of the simulation objects are offered by the DyMeLoR library [25, 27], which has been integrated with the new memory management based on stocks. In particular, the memory segments from which DyMeLoR takes the chunks to be delivered for dynamic memory allocation by a given simulation object—and keeps the corresponding map for log/restore operations—correspond to the stocks of memory destined for that object usage. We recall that relying on DyMeLoR as actual memory allocator is fully transparent also by the side of memory management API since the application programmer is still allowed to use the classical ANSI/ISO-C memory allocation functions. In fact, in the ROOT-Sim architecture, these are transparently redirected to functions internal to DyMeLoR.

As for the (temporary) binding of simulation objects to threads, we still relied on the policies already supported in ROOT-Sim [22], which are aimed at optimizing the sharing of the overall simulation workload across the threads operating in the PDES system along the whole lifetime of the simulation run. The simulation objects currently bound to a given worker thread are dispatched according to the Lowest Timestamp First policy. However, when running with ECS, the simulation objects that are in the block state are not considered in the dispatching process (thus being again eligible for dispatching only after exiting the block state). In any case, the blocking of an object does not imply blocking the activities of the thread taking care of it, since this thread is allowed to dispatch other objects and/or to run platform-level tasks.

We have run experiments on a 32-core HP ProLiant server equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. Each processor has 8 physical cores that share a 12 MB L3 cache (6MB per each 4-cores set), and each core has a 512KB private L2 cache. The machine is equipped with 64 GB of RAM—organized in 8 NUMA nodes—and runs Linux 3.2. A ROOT-Sim configuration with 32 symmetric worker threads has been used in all the experiments, with GVT and fossil collection taking place each one second. We have not used more than 32 worker threads (so we have never exceeded the total number of physical cores) to avoid interference effects which could impair performance [28]. All the models used for the experimentation are available online in the ROOT-Sim repository.

4.2. Experimental Data

This section is divided in two parts. We initially provide data for an evaluation of the overhead due to the core memory management facilities underlying cross-state events and ECS. To this end, we use a simulation model of a Personal Communication System (PCS) natively entailing disjointness of memory accesses by the different simulation objects—thus not exploiting at all ECS facilities, although paying the cost for the associated memory management support—which has been parameterized so as to achieve various runtime dynamics. After, we present data related to the assessment of the whole innovative approach to PDES—based on cross-state events—by relying on a multi-robot exploration model. In this part of the study, we also compare the runtime behavior of models exploiting ECS (implemented by relying on cross-state events) to the counterpart exclusively based on traditional PDES programming (only relying on event-dependencies via simulation objects' cross-event scheduling).

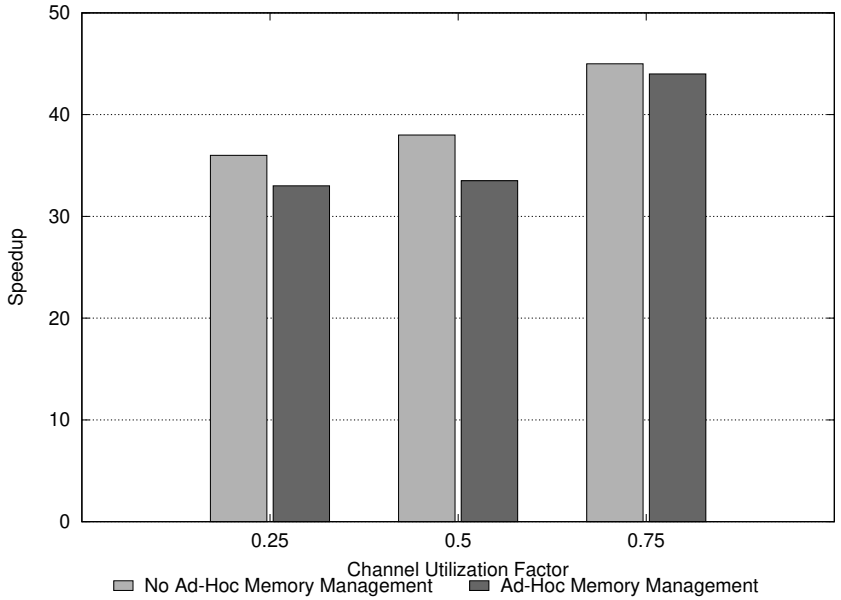

Figure 8: Relative speed-up by the parallel run without and with ad-hoc memory management vs the sequential run.

4.2.1. Pure Overhead Assessment

We evaluated the overhead induced by the memory management architecture supporting cross-state events by relying on a PCS model with 1024 wireless cells covering a square region, each one managing up to 1000 wireless channels. It models interference across different channels within a same cell and power management upon call setup/handoff in a high fidelity fashion according to the specification provided in [29]. This same model has been already used to assess the multi-threaded version of ROOT-Sim where we have integrated ECS, and its detailed description can be found in [30][5]. Two specific aspects are relevant for this study: 1) each simulation object models an individual cell, and the interactions between objects exclusively take place via handoff events of mobile devices across different cells—hence, as hinted before, memory accesses while processing the events at the different simulation objects are intrinsically disjoint; 2) the average granularity (CPU requirement) of the events is directly proportional to the wireless channel utilization factor, since the more channels are busy, the more complex is the calculation of interference and Signal-to-Interference Ratio (SIR) while simulating power regulation.

We have run this model with three different configurations for the channel utilization factor, namely 25%, 50% and 75%, which gave rise to average granularity of the simulation events ranging from the order of 30 to 100 microseconds on the used computing platform. Also, we considered three different execution settings: a classical sequential execution (based on a calendar queue event scheduler), a parallel execution where no ad-hoc memory management facility is activated, and a parallel execution where we rely on the innovative memory

[5]The source code is anyhow available within the ROOT-Sim models' repository.

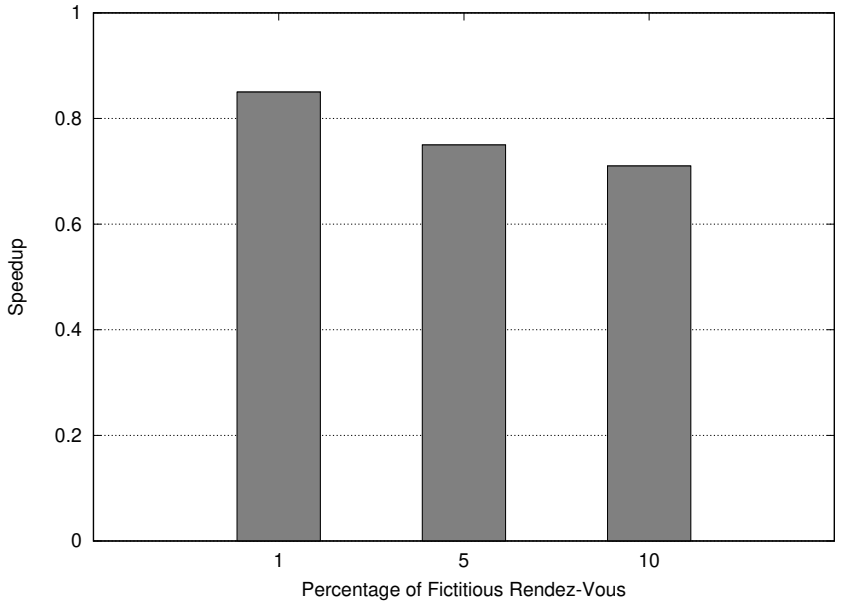

Figure 9: Relative speed-up by the ad-hoc memory management architecture vs the classical parallel run.

management architecture needed to support cross-state events and ECS. The latter setting entails switching between object-mode and platform-mode at each processed event—because of the need for managing the access to different memory stocks—with refill of the `CR3` register and implicit squash of the TLB. This setting allows us to assess the pure mode-switch overhead operated by the support for ECS as evaluated in a scenario where ECS is useless since no cross-state event is ever detected in the native simulation model implementation.

In Figure 8, we show the variation of the speedup (vs the sequential run) we observed for simulating on the order of 1 million (committed) events in the different parallel execution modes (each sample is the average over 10 runs based on different random-generation seeds). By the data, we see how the maximal loss in performance by the ad-hoc memory management architecture switching between platform and object modes is on the order of 9% and is observed for the case of finer grain simulation events—namely, for the case of 25% utilization factor. Such a performance penalty almost disappears for coarser grain configurations.

Then, we modified the PCS model in order to generate fictitious cross-state events periodically. When one fictitious cross-state event occurs, the executing object simply performs a dummy read operation into the state of an adjacent cell. However, we do not really enable ECS synchronization, as the dummy read access is not important for the correctness of the simulation and can be executed out of timestamp order. Therefore, we only trap the access and open the stock associated with the object hit by the read operation. In this way we are able to assess the overhead induced by the identification at runtime of a cross-state event, when also including the management of memory faults and the activation of the ECS handler. For this experiment, we considered the PCS configuration with wireless channel utilization factor set to 50%, and we varied the frequency of occurrence of the fictitious cross-state events between 1% and 10% of the total number of events processed. In

Figure 9, we show the relative speedup achieved by the configuration with ad-hoc memory management and fault handling upon the occurrence of fictitious cross-state events vs the configuration with no ad-hoc memory management. By the results, the ad-hoc architecture induces a speed-down that, as expected, increases with respect to the frequency of fictitious cross-state events. Note that the speed-down is not only caused by the overhead for handling the memory faults. It is also due to the switch between platform and object modes, which is mandatory in order to create the per-thread memory view needed to trap the access to the state of other simulation objects. However, the speed-down is quite limited for relatively infrequent fictitious cross-state events, and becomes non-negligible only when moving towards scenarios with relatively frequent cross-state occurrences (10% of the processed events).

On the other hand, the whole ad-hoc memory management architecture has been designed and realized to provide, transparently to the application code, a unique innovative support to handle cross-state dependencies in presence of concurrent objects. Hence the loss of performance in contexts where the model to be executed exhibits intrinsically disjoint accesses across different object-states (such as for the configurations in Figure 8) is the unavoidable price to be paid for the achievement of a runtime environment offering the above-mentioned level of transparency.

4.2.2. Effectiveness Assessment

To assess programmability and performance advantages provided by cross-state events and ECS synchronization, we have implemented a multi-robot exploration and mapping simulation model, according to the results in [31]. Specifically, a group of robots is set out into an unknown space, with the goal of fully exploring it. The map is constructed during the exploration, relying on the notion of *exploration frontier*. By keeping a representation of the explored world, the robot is able to detect which is the closest unexplored area it can reach, computes the fastest way to reach it and continues the exploration.

The robots explore independently of each other until one coincidentally detects another robot. Whenever two robots enter a proximity region, they perform three different actions:

1. they use their sensors to estimate their mutual physical position—recall that they are just in *proximity*;
2. they verify the goodness of their position hypothesis by creating a meeting point in the explored part of the region, and trying to meet again there;
3. if the hypothesis is verified, they exchange the data acquired during the exploration, thus reducing the exploration time and allowing for a more accurate decision of the actions to be taken.

If step 2 succeeds (i.e., the robots actually meet again), it means that the estimation of their respective position is correct. Therefore, they can form a *cluster* and start exploring the environment in a collaborative way. Specifically, this collaborative exploration can take place in two different ways. On the one hand, they jointly define (by relying on *cost* and *utility* functions, as defined in [31]) their next exploration targets, so that they can minimize the time required for a complete environment exploration. On the other hand, they might decide to make a *guess* about the position of other robots (the total number of which is

known) which are not part of the cluster yet. In the latter case, one of the robots (the one for which the utility/cost ratio is convenient) targets the hypothesized position. If a robot is found there, the aforementioned steps are carried out, so as to increase the knowledge of the environment.

When implementing a PDES simulation model for this scenario, three main hindrances are found. First, discovering the presence of a nearby robot can be difficult. In fact, either the robots must communicate to each other their current position—thus exponentially increasing the number of exchanged messages (events) which in turn can limit the performance of the simulation—or they have to notify it to specific simulation objects (i.e., the regions), again increasing the number of messages/events exchanged. Second, to estimate the respective position of the agents, various simulation events could be required. Third, exchanging map information could entail a data transfer non-negligible in size, posing a huge burden on the communication subsystem. Additionally, all these programmatic steps are not straightforward, as they force the modeler to reason according to the state-separation paradigm proper of PDES.

On the other hand, relying on cross-state events and ECS allows for a completely-transparent synchronization of the simulation objects involved in any mutual state update, which therefore simplifies the development process of the simulation model. In our implementation we rely on two different types of simulation objects, namely active ones (implementing the robots) and passive ones (implementing regions of the explored environment). More specifically, the environment is represented as a square region, divided into hexagonal cells. This choice allows us to define a meaningful mobility model for the agents, and at the same time allows us to define proximity regions which are used by the agents to detect the presence of other robots in the nearby. Also, periodic events occurring into any cell are envisaged as the basis for modeling the evolution (inside the cell) of any phenomenon characterizing the dynamic change in the state of the explored region.

At simulation startup, each passive simulation object creates random obstacles (which prevent the agents from reaching any neighbor cell), mimicking a rescue scenario where an open space is modified by an accident and the robots are used to explore it for rescue activities. At the same time, each passive object instantiates in its private simulation state (by relying on a standard call to `malloc()`) a *presence bitmap*. Each bit is associated with a specific robot, and its value is associated with the robot being in the cell or not. By relying on a fast bitmap scan, each robot (thus each active simulation object) is able to discover which robots are present in the cell. Finally, the passive object registers its simulation state by storing a pointer in a global array called `states[]`, thus allowing any other simulation object to directly access (and/or modify) it. This is done by relying on a code block where the modeler is not required to rely on any platform-specific API, as illustrated in the following code snippet which is executed upon initializing the simulation model.

```
1 // Allocated state
2 state = malloc(sizeof(agent_state_type));
3
4 for(i = 0; i < 6; i++) {
5         if(isValidNeighbour(me, i)) {
6                 // With a random probability, an obstacle prevents from getting there
7                 if(Random() < OBSTACLE_PROB) {
```

```
8                       state->neighbours[i] = -1;
9               } else {
10                      state->neighbours[i] = GetNeighbourId(me, i);
11              }
12      } else {
13              state->neighbours[i] = -1;
14      }
15 }
16 // Allocate the presence bitmap
17 state->agents = malloc(BITMAP_SIZE(n_prc_tot - num_cells));
18 bzero(state->agents, BITMAP_SIZE(n_prc_tot - num_cells));
19 // Register the state in a global array. This is forbidden in traditional PDES
20 states[me] = state;
```

`isValidNeighbour()` tells whether a cell is placed on the boundary of the square region, `GetNeighbourId()` performs hexagonal-to-linear coordinates conversion for detecting a neighbour id, `num_cells` is a variable defined by the model (and initialized at simulation startup by the user) which tells how many cells must be used to represent the square region, `BITMAP_SIZE` is a model-defined macro which converts the number of agents to be represented into the bitmap to a number of sequential bytes providing the relative number of bits, `n_prc_tot` is a variable initialized at simulation startup which tells the total number of simulation objects in the current run, and `me` is a unique integer used to identify a specific simulation object, passed as input from the ROOT-Sim runtime environment to callbacks giving control to the application for event processing (as in most traditional simulation frameworks).

Thanks to cross-state events and ECS, when an agent enters a hexagonal cell (the id of the cell is piggy-backed on the event), registering its presence in the cell (which corresponds to a different concurrent object) is as simple as:

```
21 state->current_cell = event_content->cell;
22
23 // Register the position of the robot in the cell. This is inconsistent in traditional PDES
24 cell = (cell_state_type *)states[state->current_cell];
25 cell->present_agents++;
26 SET_BIT(cell->agents, me - num_cells);
```

Then, the agent has to collect information about the environment—which in our model is represented by the obstacles in the current cell—and has to detect the presence of additional robots. Since we have direct access to the cell's state thanks to cross-state events and ECS, this can be easily done by reading this information from it:

```
27 // Mark the cell as explored and "discover" the surroundings. This is inconsistent in traditional PDES
28 state->visit_map[state->current_cell].visited = true;
29 memcpy(&state->visit_map[state->current_cell].neighbours, cell->neighbours, sizeof(unsigned int) * 6);
30
31 // Is there any other robot in the cell?
32 if(cell->present_agents > 1) {
33         <scan the bitmap>
34 }
```

In case one robot is found in the cell, then the agent simply "merges" its view of the environment. This can be easily done by relying again on the `states[]` array, provided that each robot registers a pointer to its private state in it at simulation startup:

```
35 robot = (agent_state_type *)states[robot_index];
```

```
36 for(j = 0; j < num_cells; j++) {
37         if(robot->visit_map[j].visited) {
38                 memcpy(&state->visit_map[j], &robot->visit_map[j], sizeof(map_t));
39         } else if (state->visit_map[j].visited) {
40                 memcpy(&robot->visit_map[j], &state->visit_map[j], sizeof(map_t));
41         }
42 }
```

No notion of parallelism is present in the shown code snippets from the simulation model, yet (by relying on cross-state events and ECS) the ROOT-Sim platforms is able to run the simulation model in parallel. We emphasize that the simulation model is written in an easy-to-manage style, without the need for respecting any PDES-related constraint, except for the signature of the event handlers, which is compliant with the one adopted by the ROOT-Sim platform.

To provide more details, ECS is triggered in different lines of the above code snippets. In particular, lines 25 and 32 trigger the synchronization with a cell (i.e., the current cell hosting the robot), while lines 37, 38, and 40 synchronize with another robot, namely one of the (possibly multiple) robots found to be in the cell. It is interesting to note that lines 25 and 32 access the state in different ways, the former being a read/write operation, the latter being simply a read operation. Similarly, line 37 accesses the state in read mode, while lines 38 and 40 entail writing multiple bytes of memory. Nevertheless, despite the multiple accesses, ECS requires executing the synchronization protocol (via rendez-vous events) only once per each other simulation object the state of which is being accessed while processing the current event.

As for runtime performance, we simulated a region of 4096 cells, and we changed the number of robot units moving around to 100, 500, and 1000. Higher values (500 and 1000 agents) give rise to average agent density into the explored space on the order of 0.12 and 0.24 per cell, respectively. Although these values might be above realistic settings, especially for cells modeling regions with non-minimal size, they help assessing how the performance of the ECS support for easy of programming and transparent parallelization scales vs variations of the ratio between the number of passive simulation objects (the cells) and the number of active ones (the agents). On the other hand, the configuration with 100 agents gives rise to an average density of per-cell agents of 0.02, realistically representing cases where, e.g., a reduced number of highly-specialized agents is employed for the exploration of a non-minimally sized region.

In Figure 10, we report the execution time of the simulation model for the three different cases, and for different amounts of ROOT-Sim worker threads (8, 16, and 32) deployed on top of the used multi-core computing system. We also report the execution time for running the very same application code sequentially, again on top of a classical calendar-queue scheduler. By the data we see how the parallel runs provide speedup that ranges between 30 and 35, which also linearly scales while varying the number of worker threads in the parallel simulation platform. This demonstrates the effectiveness of ECS in delivering adequate parallel performance, beyond providing supports for easy of programming. Also, the implementation patterns used in our model can constitute a reference for different scenarios entailing both passive (region) and active (agent) simulation objects.

Finally, in Figure 11, we report performance data for a comparison between the above described ECS-based implementation of the simulation model, and one that exclusively relies on the classical PDES paradigm only performing fully disjoint accesses to the object states by the threads. In this variant, information sharing across the different objects while the simulation run proceeds is only based on events' exchange—the information to share at a given simulation time is piggybacked on the timestamped events as their payload. These exchanges take place whenever two (or more) robots meet in the map, and they have to merge their knowledge. In particular, every robot sends to any other robot in the just-formed cluster an event piggybacking the information associated with the whole portion of the environment that has been explored so far. Upon processing such a message, a robot scans the payload and installs in its own private state all the information which is not present yet. In the plots, we refer to this implementation of the model as TR (Traditional), as opposed to the ECS-based one. For this comparative study we also vary the number of robots exploring the target area in the simulation ('R' in the plots) and the number of threads used to carry out the parallel runs. The reported data points still come from an average over 10 different runs, all executed with different random-generation seeds.

By the results, we can see that independently of the number of robots and the number of active worker threads, ECS-based synchronization provides a performance increase with respect to the traditional PDES run, which ranges from 19% to 58%. Anyhow, a more notable aspect is that when the number of robots increases, a more parallel execution—with increased numbers of worker threads—gives rise to a higher gap between ECS-based and traditional PDES. These phenomena are related to the fact that the event-exchange and the memory copy costs have a more prominent effect that is adverse to traditional PDES. On the other hand, ECS-based synchronization better scales with respect to both the model size and the used computing system power (number of CPU-cores). It is also important to emphasize that the variant of the simulation model based on traditional PDES requires much more code to be written by the simulation model's developer, which has also to manually implement some form of marshalling of the simulation state, in order to transmit its content as an event payload using the API of the underlying runtime environment. To provide a quantification of the benefits on programmability, we have that the ECS-based version of code has 25% less lines of code with respect to the traditional version not exploiting cross-state events[6].

By the whole experimentation, we can conclude that cross-state events, and the associated ECS synchronization protocol, have the twofold benefit of delivering a non-negligible performance increase, anytime that the amount of data to be accessed across two objects is non-minimal, while enabling in any case a much simpler programming model.

5. Related Work

Several studies in the literature have been aimed at coping with the issue of state disjointness across concurrent objects in PDES systems, so as to allow some form of data (hence

[6]Line counting refers to the versions of the two different models implementations available in the "models" branch of the ROOT-Sim respository.

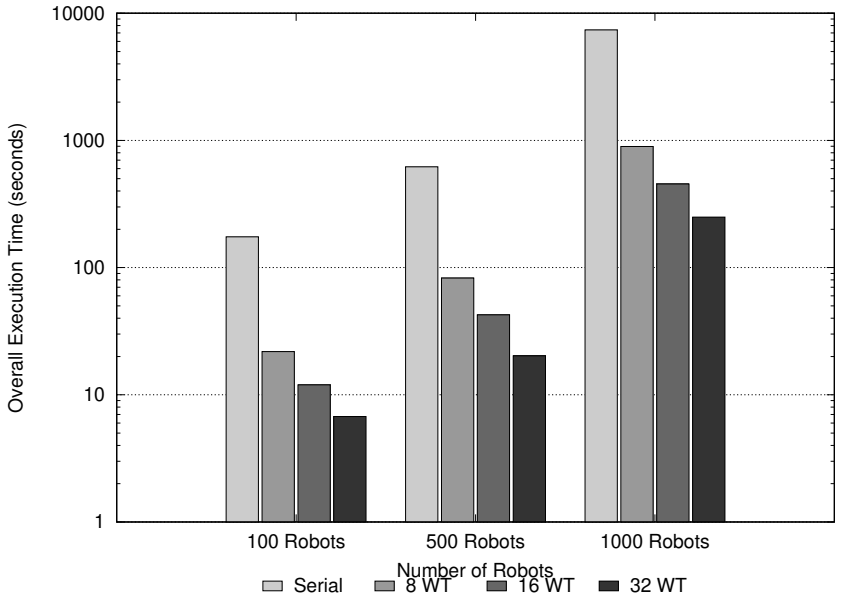

Figure 10: Sequential vs parallel ECS-based execution times (**log-scale** on y-axis).

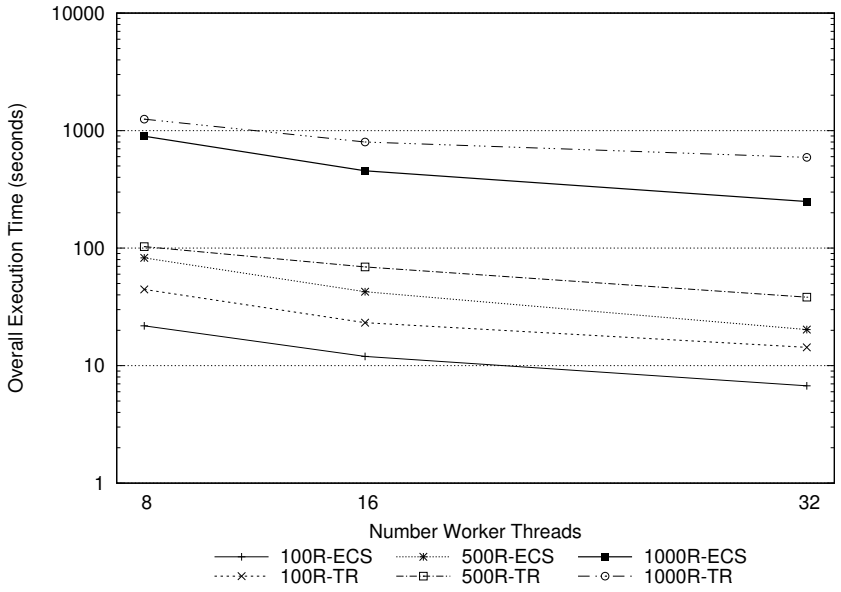

Figure 11: Traditional PDES vs parallel ECS-based execution times (**log-scale** on y-axis).

state) sharing. The work in [32] discusses how state sharing might be emulated by using a separate simulation object hosting the shared data and acting as a centralized server. This proposal also introduces the notion of *version records*, where multi-versioning is used for shared data maintenance in order to cope with read/write operations occurring at different logical times, and to avoid unneeded rollbacks of the centralized server in case of optimistic synchronization. This is an approach similar to the one proposed in [6], where a theoretical presentation of algorithms to implement a Distributed Shared Memory mechanism is provided in terms of protocols to keep replicated instances of a variable coherent. In particular, one of the provided algorithms proposes to implement variables as multi-version lists where write operations install new version nodes and read operations find the most suitable version. The above approaches are different from what we propose given that read/write access to shared variables is mapped to message-passing (namely, event schedule operations), while we support in-place access to any (by default sharable) buffer within the simulation object states, e.g., via pointers. Also, in our proposal sharing is not limited to a particular memory slice (such as the state image of the centralized server), while we allow access, and hence sharing, of any memory buffer representing a portion of the whole simulation model state. Also, by design the above approaches are strongly oriented to distributed simulation environments, while we target the trend of shared-memory/multi-core machines.

In [33], the notion of *state query* is introduced, where a simulation object needing the value of a portion of the state that belongs to a different object can issue a query message to it and then waits for a reply containing the suitable value. If this value is later detected to be no longer valid, an anti-message is sent so as to invalidate the query. Again, this approach relies on message passing, and is not transparent to the application programmer, who needs to embed the usage of query messages within the application code.

The work in [34] proposes to integrate the support for shared state in terms of global variables, by basing the architecture on [35]. Although this proposal supports in-place read/write operations as we do (i.e., simulation objects directly access the only copy of the data, avoiding a commit phase at the end of the execution of an event), it provides no transparency, as the application-level code must explicitly register a simulation object as a reader/writer on shared variables. Our proposal avoids this limitation by also allowing the sharing of dynamically-allocated buffers, for which pre-declaration of the potential need to access cannot be raised at startup (hence intrinsically leading actual access to be determined as a function of the specific execution trajectory while running the application).

The issue of transparency has been tackled in [36], where shared data are allowed to be accessed by concurrent objects without the need for pre-declaring the intention to access. This has been achieved via user transparent software instrumentation, in combination with a multi-version scheme, either allowing the redirection of read operations to the correct version of the data (on the basis of the timestamp) or forcing rollbacks of causally inconsistent reads. This solutions is targeted at the management of global variables. Instead, our proposal is suited for data sharing of dynamically allocated memory chunks logically incorporated within the state of each individual simulation object, while still providing parallelism and synchronization transparency.

In the context of the High-Level-Architecture (HLA), proposals for supporting shared-

state can be found in [7, 37]. They are again targeted at distributed environments, since they are based on a middleware component which relies on a timestamp-ordering approach for implementing a request/reply protocol. Additionally, these approaches are targeted at the conservative synchronization protocol, where there is no need to detect and handle causality violations, while we target optimistic synchronization.

The work in [16] proposes a framework targeted at multi-core machines and based on Time Warp, where so called Extended Logical Processes (Ex-LP), defined as a collection of LPs, have public attributes that are associated with variables which can be accessed by LPs in other Ex-LPs. The work proposes to handle the accesses to shared attributes by relying on a specifically targeted Transactional Memory (TM) implementation, where events are mapped to transactions and the actual implementation of the TM is based on [34]. One core difference between our proposal and the one in [16] is that the latter requires a-priori knowledge of the attributes to be shared, which need to be a-priori mapped to TM managed memory locations. This also requires explicit annotations within the application code. Rather, our proposal allows for sharing any memory area within the heap, without the need for a-priori knowledge of whether some sharing on a specific area can occur. This increases the level of transparency. In fact, the programmer is allowed to let any simulation object that takes control touch any valid memory location within the global simulation state without the need for any particular care. Overall, we "transactify" the access to memory chunks across different concurrent objects without the need for marking data portions subject to transactional management by the programmer.

6. Conclusions

We have presented a new approach to PDES based on the introduction of cross-state events. These events can access the state of multiple concurrent simulation objects, thus being particularly useful in contexts where the PDES application is based on sharing information across the objects along the model execution. In the traditional PDES paradigm, only entailing disjoint accesses to the simulation object states, such a sharing would need to be supported via the exchange of events with proper payloads. This not only hampers performance, as we have shown in the experimental assessment of our new proposal, but rather can also impact the application programming complexity—this is due to the need for coding the calls to some event-scheduling API and the packing/unnpacking of event payloads. On the contrary, we offer a runtime support for cross-state events, which also enables speculative event processing, such that the programmer does not need to use any specific API or programming rule in order to implement cross-state events—namely, to directly share information across concurrent simulation objects—in his application. He can simply code the application modules letting them perform in-place accesses (e.g. pointer-based accesses) to whatever simulation object state. This is a fully new solution compared to any literature approach devoted to the study of data/information sharing in PDES.

Overall, our protocol for runtime handling of cross-state events, which we called ECS (Event and Cross-State) synchronization, allows breaking the classical limit of PDES, where any event processing handler can access memory limited to the local state of an individual

simulation object. Rather, with ECS the application is allowed to touch (in either read or write mode) any valid memory location currently representing any portion of the overall simulation model state. This capability has been achieved thanks to the design and implementation of an innovative memory management architecture, which creates per-thread views of memory protection and tracks memory accesses towards specific simulation object states, for transparent handling by ECS, in an efficient manner. Our proposal targets PDES platforms to be run of top of shared memory multi-core machines, which nowadays represent mainstream (off-the-shelf) architectures. We again emphasize that cross-state events are transparently handled at runtime in our architecture, which jointly enables concurrency and speculative processing of events across the simulation objects as in classical speculative PDES.

References

[1] R. M. Fujimoto, Performance of Time Warp Under Synthetic Workloads, in: Proceedings of the Multiconference on Distributed Simulation, Society for Computer Simulation, 23–28, 1990.

[2] E. Santini, M. Ianni, A. Pellegrini, F. Quaglia, Hardware-Transactional-Memory Based Speculative Parallel Discrete Event Simulation of Very Fine Grain Models, in: 22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015, 145–154, 2015.

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

[4] R. Vitali, A. Pellegrini, F. Quaglia, Towards Symmetric Multi-threaded Optimistic Simulation Kernels, in: 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation, PADS, IEEE Computer Society, ISBN 978-1-4673-1797-9, ISSN 1087-4097, 211–220, doi:10.1109/PADS.2012.46, URL `http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6305914`, 2012.

[5] R. Vitali, A. Pellegrini, F. Quaglia, Load sharing for optimistic parallel simulations on multi core machines, SIGMETRICS Performance Evaluation Review 40 (3) (2012) 2–11, doi:10.1145/2425248.2425250, URL `http://doi.acm.org/10.1145/2425248.2425250`.

[6] H. Mehl, S. Hammes, How to integrate shared variables in distributed simulation, SIGSIM Simulation Digest 25 (2) (1995) 14–41, ISSN 0163-6103, doi:10.1145/233498.233499.

[7] B. P. Gan, M. Y. H. Low, J. Wei, X. Wang, S. J. Turner, W. Cai, Synchronization and management of shared state in {HLA}-based distributed simulation, in: Proceedings of the Winter Simulation Conference, 847–854, doi:10.1109/WSC.2003.1261503, 2003.

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

[9] B. R. Preiss, W. M. Loucks, D. MacIntyre, Effects of the Checkpoint Interval on Time and Space in Time Warp, ACM Transactions on Modeling and Computer Simulation 4 (3) (1994) 223–253.

[10] F. Quaglia, A Cost Model for Selecting Checkpoint Positions in Time Warp Parallel Simulation, IEEE Transactions on Parallel and Distributed Systems 12 (4) (2001) 346–362.

[11] F. Quaglia, A. Santoro, Non-Blocking Checkpointing for Optimistic Parallel Simulation: Description and an Implementation, IEEE Transactions on Parallel and Distributed Systems 14 (6) (2003) 593–610.

[12] C. D. Carothers, K. S. Perumalla, R. M. Fujimoto, Efficient optimistic parallel simulations using reverse computation, ACM Transactions on Modeling and Computer Simulation 9 (3) (1999) 224–253.

[13] D. Cingolani, A. Pellegrini, F. Quaglia, Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES, ACM Trans. Model. Comput. Simul. 27 (2) (2017) 11:1–11:26.

[14] F. Antonacci, A. Pellegrini, F. Quaglia, Consistent and Efficient Output-streams Management in Optimistic Simulation Platforms, in: Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, ACM, ISBN 978-1-4503-1920-1, 315–326, doi:10.1145/2486092.2486133, 2013.

[15] D. Cucuzzo, S. D'Alessio, F. Quaglia, P. Romano, A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation, in: Proceedings - IEEE International Symposium on Distributed Simulation and Real-Time Applications, DS-RT, ISBN 0769530117, ISSN 15506525, 227–234, doi:10.1109/DS-RT.2007.9, 2007.

[16] L.-l. Chen, Y.-s. Lu, Y.-P. Yao, S.-l. Peng, L.-d. Wu, A Well-Balanced Time Warp System on Multi-Core Environments, in: Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation, PADS, IEEE Computer Society, ISBN 978-1-4577-1363-7, 1–9, doi:10.1109/PADS.2011.5936752, 2011.

[17] D. Jagtap, N. Abu-Ghazaleh, D. Ponomarev, Optimization of Parallel Discrete Event Simulator for Multi-core Systems, in: Proceedings of the International Parallel and Distributed Processing Symposium, IEEE Computer Society, ISSN 1530-2075, 520–531, doi:10.1109/IPDPS.2012.55, 2012.

[18] R. Vitali, A. Pellegrini, F. Quaglia, A load-sharing architecture for high performance optimistic simulations on multi-core machines, 2012 19th International Conference on High Performance Computing, HiPC 2012 doi:10.1109/HiPC.2012.6507510.

[19] C. D. Carothers, D. W. Bauer, S. Pearce, ROSS: A high-performance, low-memory, modular Time Warp system, J. Parallel Distrib. Comput. 62 (11) (2002) 1648–1669, doi:10.1016/S0743-7315(02)00004-7, URL http://dx.doi.org/10.1016/S0743-7315(02)00004-7.

[20] J. Wang, D. Jagtap, N. B. Abu-Ghazaleh, D. Ponomarev, Parallel Discrete Event Simulation for Multi-Core Systems: Analysis and Optimization, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1574–1584, doi:10.1109/TPDS.2013.193, URL http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.193.

[21] S. Meraji, W. Zhang, C. Tropper, A Multi-State Q-Learning Approach for the Dynamic Load Balancing of Time Warp, in: Principles of Advanced and Distributed Simulation (PADS), ISBN 978-1-4244-7292-5, 1–8, 2010.

[22] R. Vitali, A. Pellegrini, F. Quaglia, Load sharing for optimistic parallel simulations on multi core machines, ACM SIGMETRICS Performance Evaluation Review 40 (3) (2012) 2, ISSN 01635999, doi:10.1145/2425248.2425250, URL http://dl.acm.org/citation.cfm?doid=2425248.2425250.

[23] A. Pellegrini, F. Quaglia, NUMA Time Warp, in: Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation, London, United Kingdom, June 10 - 12, 2015, 59–70, 2015.

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

[25] A. Pellegrini, R. Vitali, 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, IEEE, ISBN 9780769537139, 45–53, doi:10.1109/PADS.2009.24, URL http://dl.acm.org/citation.cfm?id=1577959.1577973, 2009.

[26] A. Pellegrini, F. Quaglia, The ROme OpTimistic Simulator: A tutorial, in: D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Constan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, J. Weidendorfer (Eds.), Proceedings of the Euro-Par 2013: Parallel Processing Workshops, PADABS, LNCS, Springer-Verlag, 501–512, 2014.

[27] R. Toccaceli, F. Quaglia, DyMeLoR: Dynamic Memory Logger and Restorer Library for Optimistic Simulation Objects with Generic Memory Layout, in: Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation, IEEE Computer Society, ISBN 978-0-7695-3159-5, 163–172, doi:http://dx.doi.org/10.1109/PADS.2008.23, 2008.

[28] J. Wang, N. Abu-Ghazaleh, D. Ponomarev, AIR: Application-Level Interference Resilience for PDES on Multicore Systems, ACM Trans. Model. Comput. Simul. 25 (3) (2015) 19:1—-19:25, ISSN 1049-3301, doi:10.1145/2701420, URL http://doi.acm.org/10.1145/2701420.

[29] S. Kandukuri, S. Boyd, Optimal Power Control in Interference-Limited Fading Wireless Channels with Outage-Probability Specifications, IEEE Transactions on Wireless Communications 1 (1) (2002) 46–55.

[30] A. Pellegrini, R. Vitali, F. Quaglia, Autonomic State Management for Optimistic Simulation Platforms, IEEE Trans. Parallel Distrib. Syst. 26 (6) (2015) 1560–1569.

[31] D. Fox, J. Ko, K. Konolige, B. Limketkai, D. Schulz, B. Stewart, Distributed Multirobot Exploration and Mapping, Proceedings of the IEEE 94 (7) (2006) 1325–1339, ISSN 0018-9219, doi: 10.1109/JPROC.2006.876927.

[32] D. Bruce, The treatment of state in optimistic systems, SIGSIM Simul. Dig. 25 (1) (1995) 40–49, ISSN 0163-6103, doi:10.1145/214283.214297, URL http://doi.acm.org/10.1145/214283.214297.

[33] A. Fabbri, L. Donatiello, SQTW: a mechanism for state-dependent parallel simulation. Description and experimental study, in: Proceedings of the Workshop on Parallel and Distributed Simulation, 82–89, doi:10.1109/PADS.1997.594590, 1997.

[34] K. Ghosh, R. M. Fujimoto, Parallel Discrete Event Simulation Using Space-Time Memory., in: Proceedings of the International Conference on Parallel Processing, CRC Press, 201–208, 1991.

[35] K. M. Chandy, R. Sherman, Space-time and simulation, Proceedings of the SCS Multiconference on Distributed Simulation (1989) 53–57URL http://hdl.handle.net/2060/19940004591.

[36] A. Pellegrini, R. Vitali, S. Peluso, F. Quaglia, Transparent and Efficient Shared-State Management for Optimistic Simulations on Multi-core Machines, in: Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, IEEE Computer Society, ISSN 1526-7539, 134–141, doi:10.1109/MASCOTS.2012.25, 2012.

[37] M. Y. H. Low, B. P. Gan, J. Wei, X. Wang, S. J. Turner, W. Cai, Shared State Synchronization for {HLA}-Based Distributed Simulation, Simulation 82 (8) (2006) 511–521, ISSN 0037-5497, doi: 10.1177/0037549706069342, URL http://dx.doi.org/10.1177/0037549706069342.

[38] J. I. Leivent, R. J. Watro, Mathematical Foundations of {Time Warp} Systems, ACM Transactions on Programming Languages and Systems 15 (5) (1993) 771–794, doi:10.1145/161468.161470.

[39] V. Jha, R. Bagrodia, Simultaneous Events and Lookahead in Simulation Protocols, ACM Transactions on Modeling and Computer Simulation 10 (3) (2000) 241–267, ISSN 1049-3301, doi: 10.1145/361026.361032, URL http://doi.acm.org/10.1145/361026.361032.

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

[41] R. Rönngren, R. Ayani, Adaptive Checkpointing in Time Warp, in: Proceedings of the 8th Workshop on Parallel and Distributed Simulation, Society for Computer Simulation, 110–117, 1994.

[42] R. S. Engelschall, Portable Multithreading: The Signal Stack Trick for User-space Thread Creation, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, USENIX Association, Berkeley, CA, USA, 20, doi:10.1.1.367.6776, 2000.

Appendix A. Additional Memory Management Details

Let us discuss two aspects related to the actual memory allocation support for the application code and to safety of dual-mode execution (platform vs simulation-object) in the presence of third party libraries. As for the first aspect, we integrated the memory management architecture with the DyMeLoR open source allocator [25, 27], explicitly targeting memory allocation needs in speculative PDES platforms. It intercepts dynamic memory calls by the simulation object (e.g. `malloc` calls) and handles them by managing (and delivering to the simulation object) memory chunks located into a pre-reserved memory segment. DyMeLoR also keeps bitmaps to determine the currently in use and dirty chunks, which allows to take (incremental) snapshots and restoring past states of the simulation object in case of rollbacks within the speculative processing scheme (while allowing dynamic memory to be used by the object). Integration of DyMeLoR with the currently presented architecture has been straightforward given that, rather than relying on actual `malloc` implementations for pre-reserving the segment destined to allocate the chunks for a given simulation object, in the integrated architecture we let DyMeLoR rely on the stock allocator. Hence, the virtual memory segment managed by DyMeLoR corresponds to the stock of virtual memory pages supported in the presented architecture. We note that identifying dirty chunks in DyMeLoR relies on compile/link time instrumentation of memory write instructions within the application level code. The outcoming memory access-tracking scheme is completely different from the support we are offering for cross-state events tracking, which is able to intercept read access (not only write access) to whichever application destined memory area without the need for instrumenting all the memory read instructions.

As for the second aspect raised above, DyMeLoR is shipped with wrappers for ANSI/ ISO-C stateless libraries, so that any memory allocation by these libraries (such as `strdup`) is still handled by DyMeLoR according to the above depicted scheme. Also, the third-party library interfaces are redirected to an actual logic which is statically linked to the application code, hence not requiring intervention by the dynamic linker. This automatically avoids page table updates while running in simulation-object mode which would otherwise be caused on sibling page tables by memory mapping actions by the dynamic linker (in case the shared libraries were invoked by the application code while running in simulation-object mode).

In our overall architecture we also offer re-implemented versions of core stateful libraries, such as `strtok`, by mapping their global variables onto specific stocks. This allows treating the state of the stateful library in a similar manner with respect to the state of a generic (although passive) simulation object. Hence, according to the ECS synchronization scheme, the state of the library will be accessed and manipulated in a causal-consistent manner within speculative execution (and the associated rollback scheme), depending on what events will give rise to cross-state access involving the "library object"[7].

[7]Library functions that can directly interact with non-rollbackable services of the operating system kernel, such as `printf`, need ad-hoc management approaches similar to the one proposed in, e.g., [14]. These approaches can be anyhow integrated with our software architecture given that they are based on orthogonal speculation and recoverability techniques (service interception and delay-until-commit policies for actual finalization of the service).

Appendix B. Ensuring Progress within ECS

Care must be taken to avoid deadlocks, livelocks, and the domino-effect in the rollback scheme involving rendez-vous events. Let us first consider the deadlock/livelock issue. A deadlock may arise in case of rendez-vous events cyclically involving a set of simulation objects, where the rendez-vous associated with the minimal timestamp along the cycle leads the simulation object raising this rendez-vous to wait for the rollback of a different object that is, in its turn, in the block state due to a different rendez-vous it issued, which still needs to be completed. An example situation of this type is shown in Figure B.12, where object x issues at time t_1 a rendez-vous towards object z, which is waiting for object y to reach time t_3 for a rendez-vous between z and y. On the other hand, object y is waiting for x to reach time t_2 for a rendez-vous with it.

To avoid deadlock scenarios, we can simply adopt the rule that, in case a rollback needs to be executed by a simulation object x which is currently blocked due to a rendez-vous it generated while processing an event e_x, this object is simply resumed from the block state by also squashing the finalization of the rendez-vous (this will lead to manage the rollback of the rendez-vous as explained above, e.g., by issuing the anti-event for the already sent out rendez-vous event). In other words, with our solution a rendez-vous event e with a timestamp lower than that of another rendez-vous event e' preëmpts the access to the state of the target simulation object with respect to e'. By the deadlock theory, no deadlock is possible when using preëmption.

We note that resuming from a block state implies that the current stack seen by the object also needs to be refilled with correct information (since, upon resuming, its context will no more be the processing context for the rendez-vous generating event). We note that this is a problem similar to the one of restoring the correct stack of the object upon resuming the processing of a rendez-vous generating event that lead it into the block state (so that the worker thread currently executing this object passes control to a different object, which needs to operate on a proper stack image). Details on how we handled this issue in our implementation, where the cross-state events tracking architecture and ECS have been integrated into the ROOT-Sim open source optimistic simulation platform, have been discussed in Appendix C.

We also note that annihilating the rendez-vous event via the corresponding anti-event (as discussed in Section 3.3.2) is safe even in case the destination object is currently blocked waiting for the finalization of the rendez-vous. In fact, it can be simply resumed from the block state (again with proper stack image manipulation) and can be rolled back, thus possibly altering its state image safely given that the image does no more need to be locked for the access by a different object in a rendez-vous.

We note however that unblocking the object generating a rendez-vous so as to prevent deadlock in case a rollback is required may, in its turn lead to livelock. Specifically, livelock may in principle arise in case of simultaneous events materializing circular cross-state dependencies across multiple simulation objects. Each object x along the circle, executing an event e_x at simulation time $ts(e_x)$, is hit by another object due to a cross-state memory faulting access at the same simulation time, which may lead to request the rollback of the

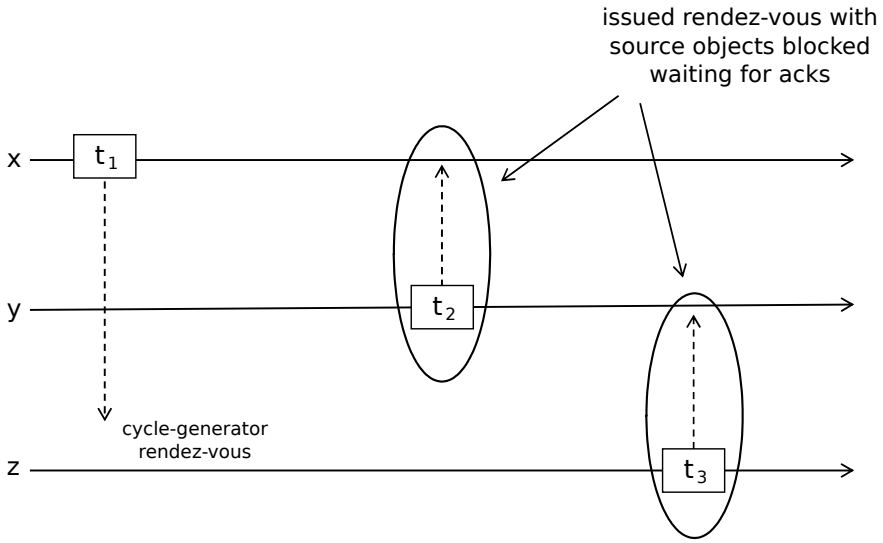

Figure B.12: Example scenario with deadlock originated by a rendez-vous generating event at time t_1 processed by simulation object x.

events generating the rendez-vous circularly. This is known to possibly lead the rollback circle to reappear indefinitely [38]. To overcome this problem, we need a priority management scheme for simultaneous events, that needs to be reflected also on the management of rendez-vous events. Particularly, if we have two events e_x and e_y such that $ts(e_x) = ts(e_y)$, and we have a priority scheme telling that $e_x \rightarrow e_y$ (namely, e_y is identified as causally dependent on e_x), then we need to enforce that any rendez-vous event e_y^{rv} generated by e_x is also causally related to e_y according to $e_y^{rv} \rightarrow e_y$. This way, the rendez-vous occurrences that are caused by events having the same timestmap are anyway sequentialized according to the priority scheme. We note however, that the guarantee of progress in (speculative) PDES systems in the presence of simultaneous events is a more general problem, with respect to what we might experience in ECS, and has been already extensively studied in literature [39]. Hence different literature solutions for tie-breaking simultaneous events (see, e.g., [40]) can be exploited for integration with ECS according to the scheme suggested above.

The final issue to cope with is the domino-effect in the rollback scheme, which as hinted is not an issue at all for the case of disjoint accesses to the simulation object states—as in traditional Time Warp. We consider the classical case where rolling back the state image of a simulation object is achieved via checkpointing—state saving— which is a widely diffused and studied scheme. In more detail, by the ample literature on log/restore in optimistic PDES systems (see, e.g., [9, 10, 41]), we know that sparse state saving, which avoids logging the simulation object state after the processing of each event, allows for optimizing the performance tradeoff between logging cost and restore cost. However, state restore at time t requires the simulation object to be rolled back to the latest state log with time less than or equal to t, and then to fictitiously reprocess intermediate events up to t in a silent mode (namely with no interactions with other objects), which is also known to as coasting-forward. In ECS this is no longer possible since a coasting-forward event

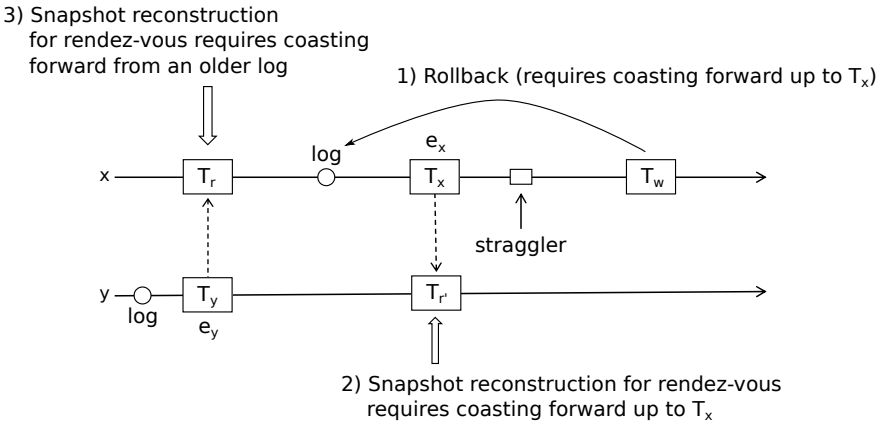

Figure B.13: Example scenario with domino-effect due to a rollback originated on simulation object x.

might be a rendez-vous generating event. Hence, in order for this to be re-processed, the simulation object originally hit by the rendez-vous also needs to rollback at the time of the rendez-vous, so as to provide its state snapshot for correct access by the object performing coasting-forward. It is easy to show that this may lead the originally rolling-back object to rollback further back along simulation time, according to the domino-effect. An example is shown in Figure B.13, where in order to execute the coasting-forward involving event e_x at object x, we need to reconstruct the snapshot of object y at time $ts(e_x)$. But this leads to the need for processing e_y in coasting forward, which in turn leads x to restore its state to a time less than $ts(e_y)$. To avoid the need for executing coasting-forwards leading to rollback interactions with other objects (thus avoiding the domino-effect), our approach is based on complementing the selected sparse state saving algorithm by forcing the log of the state of a simulation object right after the processing of a rendez-vous generating event. This will lead to the scenario where no rendez-vous generating event will ever be included in the sequence of events between two subsequent logs of the same simulation objects. Hence no rendez-vous generating event will need to be re-processed in any coasting-forward phase. On the other hand, a rendez-vous generated event also needs to be excluded by any coasting-forward, since for these events the rendez-vous source object may have performed updates into the state of the target object. To avoid a rendez-vous generated event to be included in any coasting-forward phase, we can again force a state log of the involved object right after the event is processed. These forced logs also allow coping with non-determinism of cross-state events given that no support is required for allowing them to be re-playable.

Appendix C. Management of Contexts

We discuss here the technical details related to the interruption of the execution of a cross-state event, once it is detected by our state management system. In the most general case, an *execution context* is the program state at a certain point of the lifetime of the application. This state is composed of the *CPU image* and *program variables*. The former entails all CPU registers, which allow to determine what is the next instruction to be executed (the program

counter), and which keep values commonly computed by the currently executed function, and all control registers which allow the hardware architecture and the operating system to correctly interact with the available hardware resources (e.g., MMU-related registers to correctly drive a virtual-to-physical address translation). The latter entails all the variables kept in the data sections, in the heap, and in the stack.

In our cross-state event architecture, multiple contexts should be available at the same time within the same multi-threaded application. Moreover, the multi-threaded application should be able to switch from one execution context to another at any point—namely, either when a cross-state event is detected, or when a remote simulation object is correctly realigned to a certain simulation time due to the occurrence of a cross-state event. This means that there are no "safe points" in which one context could be saved, rather the context switch might happen after the execution of any machine instruction, since memory accesses might happen everywhere, also due to some non-determinism in the application's code. The contexts we need to manage cannot therefore be associated with multiple concurrent operating system's threads. Rather, they are logically bound to a single operating system's thread, and we rely on multiple User-Level Threads (ULTs), each one associated with a simulation object, which can be activated by any worker thread—namely, at any time a worker thread can "jump" to the execution flow of any ULT. Given that simulation objects' program variables stored in the heap are transparently managed by the multi-threaded nature of the simulation engine and state management facilities (see, e.g., [25]), we rely on the following code snippet to setup a new execution context for a certain simulation object. This is freely inspired by the work in [42], although it has been carefully optimized for symmetric multithread PDES systems.

```
void context_create(LP_context_t *context, void (*entry_point)(void *), void *args, void *stack, size_t stack_size) {
     struct sigaction sa;
     struct sigaltstack ss;
     struct sigaltstack oss;

     bzero((void *)&sa, sizeof(struct sigaction));
     sa.sa_handler = context_create_trampoline;
     sa.sa_flags = SA_ONSTACK;
     sigfillset(&sa.sa_mask);
     sigdelset(&sa.sa_mask, SIGUSR1);
     sigaction(SIGUSR1, &sa, NULL);

     ss.ss_sp = stack;
     ss.ss_size = stack_size;
     ss.ss_flags = 0;
     sigaltstack(&ss, &oss);

     context_creat = context;
     context_creat_func = entry_point;
     context_creat_arg = args;
     context_called = false;
     raise(SIGUSR1);
     sigaltstack(&oss, NULL);

     context_switch_create(&context_caller, context);
}
```

The `context_create` function stores the information related to this new execution flow (namely, the CPU context and the stack's location in memory) within the `LP_context_t`

structure. To setup this new context, we rely on POSIX signals raised by the worker thread which is in charge of setting up the context for a certain simulation object. To allow the new context to live in a different stack, we rely on the POSIX-compliant `sigaltstack()` API, which asks the underlying operating system to run a signal handler within a separate stack. This stack can be allocated using any memory allocator, and is passed to the `context_create` function as an argument. Then, the `context_create` function stores a function pointer (`entry_point`) and a pointer to a vector of arguments (`args`) both passed as arguments to the `context_create` function into two TLS variables. This allows to concurrently run the creation of multiple ULTs by multiple worker threads, which allows to reduce the overhead to startup the simulation.

The worker thread then issues a call to the Posix `raise()` API, to kill itself with the `SIGUSR1` signal. Given that the `context_create` function earlier posted the `context_create_trampoline()` function as the signal handler, control is passed to it. The source of the `context_create_trampoline` function is as follows:

```
static void context_create_trampoline(int sig) {
     (void)sig;

    if(context_save(context_creat) == 0)
         return;

     context_create_boot();
}
```

The behavior of this function is quite straightforward if we correctly catch what the `context_save()` call does. This function (which we shall discuss later) is similar in spirit to the traditional Posix `setjmp()` API. In particular, the purpose of this function is to store the whole CPU state of the running thread. We recall that this signal handler is run using a different stack (the one passed via the `stack` parameter to `context_create`), so the image saved by `context_save` actually stores in `rsp` a pointer to the top of the new stack. Similarly to `setjmp`, the `context_save` function returns 0 when called directly, while it returns a user-specified value whenever the context is later restored. Since at this time we are directly calling `context_save`, the return value is 0 and the `return` statement is executed so that control returns to `context_create`. We note that this return statement completes the execution of the manually-activated signal handler, so the operating system returns the control to `context_create` in a state which is no longer related to the signal which was raised.

At this point, `context_create` relies again on `sigaltstack` to restore the previous setting (namely, new signals are not run in the previously-specified different stack) and it issues a call to `context_switch_create`, which relies on special versions of the Posix-compliant `setjmp()/longjmp()` API we have developed, and is implemented as the following macro:

```
#define context_switch_create(context_old, context_new) \
     if(set_jmp(context_old) == 0) \
        long_jmp(context_new, 1)
```

Basically, the purpose of this macro is to store the current execution context in the `context_old` variable, and to restore the context stored in `context_new`. This can be done

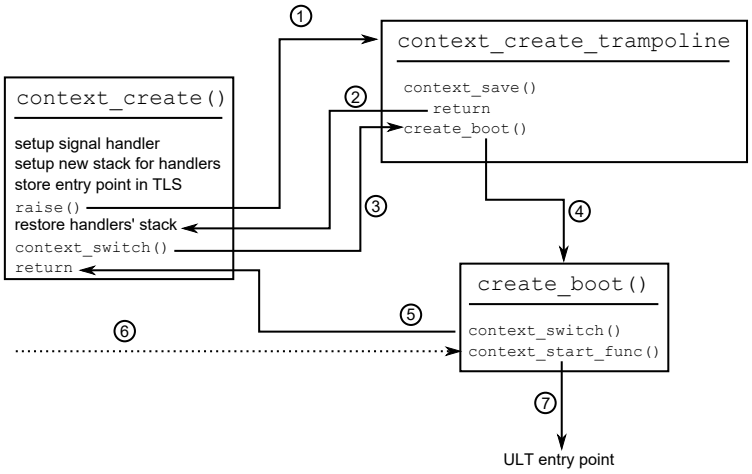

Figure C.14: Steps taken to setup a ULT—①: a signal is raised towards a signal handler using a separate stack, ②: the CPU context is saved, and control is returned to complete the signal handler, ③: context is changed explicitly, returning to the signal handler although in regular execution mode, ④: a function is called to store a local copy of the ULT entry point and arguments, ⑤: context is switched back again, to restore the original execution flow, ⑥: eventually the context is switched again, to activate the ULT, ⑦: ULT's entry point is called.

safely by checking again for the return value 0 from `set_jmp`, which is returned exactly when `context_switch_create` is referenced for the first time (i.e., we don't restore `context_new` when we return from `context_new` due to an additional context switch). Therefore, once `context_switch_create` is referenced by `context_create`, control is returned again to `context_create_trampoline`. At this time, since the second argument of `long_jmp` in `context_switch_create` is 1, the check for 0 is not satisfied, and the execution flow continues. We emphasize that this invocation to `long_jmp` already changes the stack, bringing back the stack which was setup by the operating system due to the `sigaltstack` call earlier in `context_create`. Nevertheless, the execution is not related to a signal handler now.

Next a call to `context_create_boot()` is issued. This implements the last step in the construction of the ULT context upon starting up the simulation, and is declared as a `noreturn` function. This is due to the fact that the final goal of ULTs in our system is to run simulation objects, which have a lifetime as long as the whole simulation, and therefore they are realized using a private main loop which never ends—if a simulation object has no scheduled events, then its context is simply not reactivated by the worker thread in charge of its execution. The implementation of `context_create_boot` is as follows:

```
static void context_create_boot(void) __attribute__ ((noreturn));
static void context_create_boot(void) {

     void (*context_start_func)(void *);
     void *context_start_arg;

     context_start_func = context_creat_func;
     context_start_arg = context_creat_arg;

     // Go back where the thread was created, being ready to restart from here when the ULT is scheduled!
```

```
    context_switch(context_creat, &context_caller);

    // Magically start the thread
    context_start_func(context_start_arg);

    // you should never reach this!
    abort();
}
```

By relying on it, the worker thread makes a copy of the ULT entry point and its arguments (which were temporarily stored in a TLS variable) in a couple of local variables. These local variables are actually persistent to further context changes exactly because each ULT has its own stack. At this point, everything is ready to activate the ULT, but this is postponed to the time instant the simulation object has to be activated for event processing—this will be done by calling the local function pointer `context_start_func()`. Control is now returned to the worker thread by issuing a call to `context_switch`, which restores the previous context, namely the one at the end of `context_create`. The latter function then returns, and the normal execution flow is restarted. Nevertheless, the argument `context` of `context_create` can now be used to reactivate the ULT (with a separate stack) whenever an event bound to the simulation object should be CPU-dispatched. Figure C.14 summarizes the steps which we have just described.

We shall now discuss how it is possible to explicitly switch among two different execution contexts, in our architecture. The main problem we had to face is related to context-switch management upon any possible memory-access assembly instruction generated by the compiler from the application code. To understand this issue, we must consider the fact that computing architectures rely on *calling conventions* to change the control flow between functions. Calling conventions usually divide general purpose registers between *caller-save* and *callee-save* registers. The former category encloses all the registers which, upon a function call, are not guaranteed to be saved by the called function—after the function's return point, their content might be clobbered. On the other hand, callee-save registers are those which are guaranteed to be saved by the called function before using them, so that the previous content can be restored before the function returns. It is the responsibility of compilers to coherently handle calling conventions, allowing as well for code reuse (e.g., in the context of libraries) on a same architecture. On `x86-64`, which has been used as a reference architecture for the presentation of our proposal, calling conventions are dictated by the System V AMD64 ABI. In this calling convention, callee-save registers are `rbp`, `rbx`, and `r12`–`r15`, any other general purpose register is caller-save.

Since compilers are assumed to be consistent with calling conventions, traditional implementations of `setjmp/longjmp` POSIX API functions leverage them to reduce their internal execution time. In particular, a `setjmp` is regarded by the compiler as a function call, therefore any required caller-save register is pushed before issuing that call. This allows `setjmp` to store the execution context keeping only callee-save registers `rbp`, `rbx`, `r12`–`r15`, along with the program counter `rip` and the status register `rflags`. No other register must be copied, since when control is returned after the call to `setjmp`, any required caller-save register is restored by the code generated by the compiler. Once again, we emphasize that

this (correct) behaviour is triggered by the fact that the compiler sees a function call, and enforces calling conventions.

In our scenario, such a function call is missing. In particular, we can change the flow of the program at any point, whenever a memory-access instruction targets the state of a remote simulation object. Therefore, the compiler has no clue about what registers could be clobbered, and it never emits instructions to store their value. Anyway, they might be required to be saved at any point in the program. Therefore, traditional `setjmp`/`longjmp` API simply do not work in our approach.

We therefore rely on ad-hoc versions of these functions, which are calling convention-agnostic, meaning that they save the whole CPU state. Although this is a bit more costly, it is mandatory to enforce correctness in the execution. As an additional note, x86 CPUs offer a plethora of additional registers, e.g. those related to floating-point instructions. By the definition of the calling conventions, they are all caller-save, so they must be explicitly saved/restored by our context switch facilities. We report in the following code snippet the source for the `_set_jmp()` function, which is at the heart of the implementation of the `set_jmp` which we have previously shown. It is directly implemented in assembly, due to efficiency reasons and because it explicitly breaks standard x86-64 calling conventions.

To understand the organization of this function, we note that its C-like signature is `long long _set_jmp(exec_context_t *env)`, thus it takes as its only argument a pointer to an `exec_context_t` structure, which keeps enough space to store the whole CPU state. According to System V AMD64 ABI, the first argument of a function (in case it is a pointer) is passed via the `rdi` register—when we enter the `_set_jmp` function, thus, we have already lost the possibility to save its value, so this should be handled in a different way.

```
.align 4
.globl _set_jmp
.type _set_jmp, @function
_set_jmp:
pushq %rax          # save rax, it will point to the context
pushq %r11          # save r11, it will be used as the source

lahf                # Save status flags on stack
seto %al
pushq %rax

# Save the context
movq %rdi, %rax     # rax points to the context
movq 16(%rsp), %r11 # r11 keeps the 'old' rax
movq %r11, (%rax)   # rax is the first field of the context
movq %rdx, 8(%rax)
movq %rcx, 16(%rax)
movq %rbx, 24(%rax)
movq %rsp, 32(%rax)
addq $16, 32(%rax)  # saved rsp must point one quadword above the old return address
movq %rbp, 40(%rax)
movq %rsi, 48(%rax)
movq 32(%rsp), %r11 # old 'rdi' was pushed by the surrounding macro
movq %r11, 56(%rax)
movq %r8, 64(%rax)
movq %r9, 72(%rax)
movq %r10, 80(%rax)
movq 8(%rsp), %r11  # r11 keeps the 'old' r11
movq %r11, 88(%rax) # r11 is the 12-th field of the context
```

```
movq %r12, 96(%rax)
movq %r13, 104(%rax)
movq %r14, 112(%rax)
movq %r15, 120(%rax)
movq (%rsp), %rdx   # (%rsp) is flags
movq %rdx, 136(%rax)

movq 24(%rsp), %r11 # Save the original return address
movq %r11, 128(%rax)

# Now save other registers. fxsave wants memory aligned to 16 byte.
# The context structure is aligned to 16 bytes. We have 18 8-byte
# registers, so the next address is correctly aligned.
fxsave 144(%rax)

addq $24, %rsp
xorq %rax, %rax # return 0 because the context is being created
ret
```

The idea behind our construction of this context-save facility is to move the content of all CPU registers in a memory buffer. To this end, we must use some register as a "pointer". This register is `rax`, so in order to preserve its value we first push its content to the stack. We then use `rax` to save the status register. We rely on three fast instructions rather than on the x86 `pushf` one because `pushf` has the drawback to flush the whole CPU pipeline, which is a cost which we do not want to pay at all. The trick relies in that `lahf` stores everything that is needed to restore ALU-related bits, except the signed overflow flag, which is anyhow saved in the low part of `rax` using the `seto` instruction. We can then save all registers to the buffer: this is done via a couple of `mov` instructions (we recall that the pointer to the buffer was passed via `%rdi`). Some register's content must be reconstructed, such as the stack pointer `%rsp`, since we executed within this function a couple of `push` instructions. Nevertheless, since we know the number of `push`es, we can determine the offset to apply to the current value of `%rsp`. Similarly, we clobbered the `%rax` register to save the content of the status flags, but we pushed it beforehand, so we can retrieve it from the stack. Similarly, since we issued a call to `_set_jmp`, we can retrieve the original program counter's value from the stack.

To save the remainder of the CPU state, we must save all other caller-save registers which are used to support all floating-point instructions. Manually saving them could be difficult and performance-unfair, as the number of these registers is large, and they cannot be accessed using traditional `mov` instructions. Therefore, we rely on the `fxsave` instruction, which is offered by `x86` architectures to save very quickly all registers used for floating-point and vectorized instructions.

To complete the execution of our context-save procedure, we return 0, to be compliant with the original semantic of `setjmp`—by the calling convention the return value is always stored into `rax`. Nevertheless, we still have to discuss how we can save the original value of `rdi`, which is clobbered by the function call due to calling conventions. In fact, `context_switch` does not call `_set_jmp` directly, rather it relies on the `set_jmp` macro which is defined as follows:

```
#define set_jmp(env)  ({\
    int _set_ret;\
```

```
    __asm__ __volatile__ ("pushq %rdi"); \
    _set_ret = _set_jmp(env); \
    __asm__ __volatile__ ("add $8, %rsp"); \
   _set_ret;\
})
```

This macro evaluates the return value of `_set_jmp`, making it compliant in its turn with the original `setjmp`, but before issuing the call to `_set_jmp`, it pushes the value of `rdi` on the stack (and similarly removes it from the stack after the call returns). In fact, looking at the code of `_set_jmp`, we explicitly retrieve the original value of `rdi` from the stack, since it was pushed by the surrounding macro.

To discuss how we can restore a previously-saved context, we first emphasize that the actual program counter's value that we have saved in the CPU context is the address of the first instruction after the call to `_set_jmp`, namely the instruction that stores `_set_jmp`'s return value into `_set_ret` in the macro. Therefore, to discriminate whether we are returning from a `_set_jmp` to a `longjmp` (or equivalent) we can play with this return value. Nevertheless, care must be taken to restore as well the actual original value of `rax`, which is used to store the return value. First, we introduce the C-like signature of our `_long_jmp` (which is presented in the next code snippet) as `__attribute__ ((__noreturn__)) void _long_jmp(exec_context_t *env, long long val)`, where `val` allows to specify the return value that we want to use as the return value of our "fake" invocation of `_set_jmp` upon a context restore. This value is stored into the `rsi` register, as per the calling conventions.

```
.align 4
.globl _long_jmp
.type _long_jmp, @function
_long_jmp:
movq %rdi, %rax # rax points to the context

movq 128(%rax), %r10 # This is the old return address
movq 32(%rax), %r11 # r11 is the old rsp
movq %r10, 8(%r11) # restore the old return address

movq %rsi, (%r11) # Put on the old stack the desired return value

movq 8(%rax), %rdx # rdx is the second field of the context
movq 16(%rax), %rcx
movq 24(%rax), %rbx
movq 32(%rax), %rsp
movq 40(%rax), %rbp
movq 48(%rax), %rsi
movq 64(%rax), %r8
movq 72(%rax), %r9
movq 80(%rax), %r10 # Finish to restore GP registers
movq 88(%rax), %r11
movq 96(%rax), %r12
movq 104(%rax), %r13
movq 112(%rax), %r14

# Restore FLAGS
movq 136(%rax), %rax # this is flags
addb $0x7f, %al # Overflows if OF was set
sahf

# Restore remaining rdi and r15
movq %rdi, %rax # rax now points again to context
```

```
movq 56(%rax), %rdi
movq 120(%rax), %r15

fxrstor 144(%rax) # Restore other registers

movq 32(%rax), %rsp # (possibly) change stack
popq %rax # Set the desired return value
ret # do the long jump
```

In our implementation, we play a bit with the "other" stack, namely the stack of the destination ULT. This can be done since we can extract from the stored execution context the value of `rsp` before having restored it in the current CPU state. In our code, register `r11` is used to this end. In this way, while executing on the current stack, we can read/write values to the destination stack. In fact, we use this facility to make a copy of `val` (stored in the `rsi` register) on the destination stack. This frees the `rsi` register, which can be restored along with all other registers. To move the return value from the stack to `rax`, we just issue a `pop rax` instruction just before returning from `_long_jmp`. To restore `rflags`, we use a triplet of instructions in a way similar to what we did in the context-save procedure, considering that `addb $0x7f, %al` generates an overflow only if the overflow flag was set during the context-save procedure. After this point, we must ensure that no arithmetic/ logical instructions are executed. To restore floating-point registers, we use the companion `fxrestor` instruction.

To actually perform the context switch, we use a trick similar to what we did to restore `rax`. In particular, we read from the saved context the old program counter's value, and we push that on the destination stack. Therefore, once the stack is changed, on the top of the stack we find exactly the address of the instruction next to the call to the corresponding `_set_jmp`. A `ret` instruction will "jump" to the program counter's value of the destination CPU context. In order to have the `_long_jmp` restore the actual original value of `rax`, it can be called as `_long_jmp(context_new, (context_new)->rax)`.