Porting Event & Cross-State Synchronization to the Cloud

Matteo Principe
Sapienza, University of Rome
Lockless S.r.l.
matteo.principe92@gmail.com
principe@lockless.it

Tommaso Tocci
Barcelona Supercomputing Center
tommaso.tocci@bsc.es

Alessandro Pellegrini
Sapienza, University of Rome
Lockless S.r.l.
pellegrini@diag.uniroma1.it
pellegrini@lockless.it

Francesco Quaglia
University of Rome "Tor Vergata"
Lockless S.r.l.
francesco.quaglia@uniroma2.it
quaglia@lockless.it

ABSTRACT

Along the years, Parallel Discrete Event Simulation (PDES) has been enriched with programming facilities to bypass state disjointness across the concurrent Logical Processes (LPs). New supports have been proposed, offering the programmer approaches alternative to message passing to code complex LPs' relations. Along this path we find *Event & Cross-State* (ECS), which allows writing event handlers which can perform in-place accesses to the state of any LP, by simply relying on pointers. This programming model has been shipped with a runtime support enabling concurrent speculative execution of LPs limited to shared-memory machines. In this paper, we present the design of a middleware layer that allows ECS to be ported to distributed-memory clusters of machines. A core application of our middleware is to let ECS-coded models be hosted on top of (low-cost) resources from the Cloud. Overall, ECS-coded models no longer demand for powerful shared-memory machines to execute in reasonable time. Thanks to our solution, we retain indeed the possibility to rely on the enriched ECS programming model while still enabling deployments of PDES models on convenient (Cloud-based) infrastructures. An experimental assessment of our proposal is also provided.

CCS CONCEPTS

• **Computing methodologies** → **Discrete-event simulation**; • **Theory of computation** → *Shared memory algorithms*; • **Software and its engineering** → *Distributed memory*;

ACM Reference Format:
Matteo Principe, Tommaso Tocci, Alessandro Pellegrini, and Francesco Quaglia. 2018. Porting Event & Cross-State Synchronization to the Cloud. In *SIGSIM-PADS '18: SIGSIM Principles of Advanced Discrete Simulation CD-ROM, May 23–25, 2018, Rome, Italy*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3200921.3200929

SIGSIM-PADS '18 , May 23–25, 2018, Rome, Italy

ACM ISBN 978-1-4503-5092-1/18/05…$15.00
https://doi.org/10.1145/3200921.3200929

1 INTRODUCTION

For a long time, the literature on Parallel Discrete Event Simulation (PDES) has been focused on improving the runtime behavior of PDES systems under its traditional programming model [12], where interactions between concurrent simulation objects—also known as Logical Processes (LPs)—were expressed only via event exchange. Along this path, we can find solutions oriented to load balancing [2, 4, 8, 14], to the optimization of rollback management in case of speculative processing [5, 9], and to the effectiveness of platform-level data structures and algorithms [15].

More recently, also thanks to significant changes in hardware platforms occurred since the time PDES was born, new trends of research emerged. Specifically, the possibility to rely on multi-core machines offering shared-memory support has given rise to new programming approaches for PDES, together with their transparent runtime support. These new programming approaches improve, on the one hand, the expressiveness and flexibility of model implementation while, on the other hand, they improve the execution performance when compared to explicit event scheduling to implement interactions among LPs.

Along this research path we find solutions enabling the sharing of subset of LP attributes [7], making them accessible while processing any event, or solutions oriented to let the concurrent execution of events share global data across multiple cores [17]. These proposals enable the programmer to store data produced/updated by the execution of some event in such a way that these same data can be directly accessed when later (or concurrently) processing another event—with no need for any explicit data passing at the application software level. Lines of code can be therefore reduced, together with the volume of messages that need to be exchanged at the level of the PDES platform.

A highly-flexible programming approach still based on the exploitation of shared-memory support is referred to as *Event & Cross-State* (ECS) [18]. It allows the programmer to write event handlers that can access any memory location belonging to the state of any LP via pointers. Accesses are supported in both read and write mode, thus providing a very expressive way to implement the event logic. Indeed, any event can observe the current state of the overall model or can update any of part of it. This is possible even though the runtime system manages the LPs concurrently, thus enabling the concurrent execution of multiple event handlers at the same time.

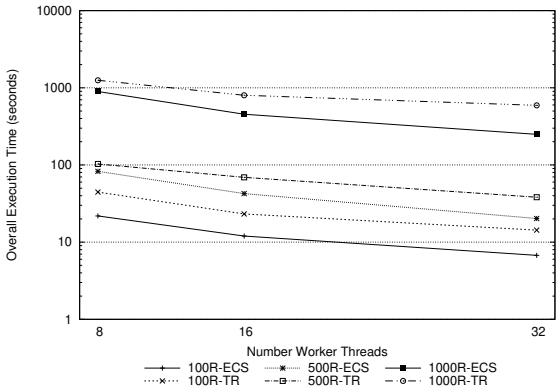

Figure 1: Traditional PDES vs Parallel ECS-based Execution Times (Log Scale on y-axis).

Correctness of read/write operations—namely causal consistency of the operations on the basis of data/timestamp dependencies—are transparently supported via the integration of both operating-system and user-space facilities. Also, ECS is conceived to work with speculative-processing runtime environments, thus enabling the exploitation of parallelism while processing independent event execution paths on multi-core systems.

To illustrate the power of ECS, we report in Figure 1 the execution time of a traditionally-coded robot exploration model (based on explicit event exchange), and of the same model coded and run on top of ECS. Both implementations have been run on the same PDES speculative platform, namely ROOT-Sim [23], with or without ECS support. The core parameter that has been varied between 100 and 1000 is the number of robots (marked as 'R'), each modeled by a different LP. The number of cells forming the region of interest, still modeled by different LPs, is set to 4096. The plot shows the execution time while varying the number of threads used to run the model between 8 and 32. All experiments are carried out on an HP ProLiant G7 machine with 32 physical cores and 64 GB of RAM. The results show that, independently of the number of robots and the number active worker threads, ECS-based runs provide a performance increase with respect to traditional PDES, which ranges from 19% to 58%. Also, event handlers in the ECS case require 25% less lines of `C` code[1]. However, one limitation of ECS is related to the fact that its runtime support targets a single shared-memory machine. Therefore, if some scale up of the computing power is required to run more demanding models in reasonable time, the user is forced to resort to a single higher-end multi-core machine.

In this paper we tackle the following question: "*can we run ECS-based models on top of clusters of low-cost resources (i.e., with limited parallelism) like spot instances from the Cloud?*". Enabling this kind of deploy would allow programmers to still access via pointers any LP state in read/write mode, and would allow end users to run large models without the need for a costly shared-memory machine. A distributed memory cluster made up by low cost (virtual) machines would in fact suffice.

[1]Models source code is available at `https://github.com/HPDCS/ROOT-Sim/` on the `models` branch.

We respond positively to such a question by providing innovative operating-system and platform-level capabilities, which make ECS a distributed middleware enabling such a seamless execution on top of distributed-memory systems. Essentially, we provide an innovative memory-management support for Linux on x86_64 systems based on new kernel-level facilities, which virtualizes a unique address space on top of a distributed memory system. At the same time, the innovative middleware facilities transparently track per-thread read/write accesses onto this address space in order to trigger the execution of middleware-level tasks. They (re)-materialize memory pages associated with the state of a simulation object at the correct simulation time on the (remote) node were the event performing the access is running. In other words, our memory-management system implements a *lease-based mechanism* where some operating system pages—and its content related to a given virtual-time instant along model execution—is granted for use to (and materialized on) a given node for a while, depending on model execution's trajectory and overall state accesses.

It is important to note that our ultimate goal is not to improve performance when running ECS-based models in the Cloud, compared to traditional PDES models run on the same Cloud platforms. Rather, our aim is to enable ECS-based programming in the Cloud, with direct benefits in terms of simplification of the programmer's job, while still guaranteeing adequate runtime performance.

Our innovative middleware has been integrated within the ROOT-Sim PDES environment [23], and is available for download. In this paper we also report experimental results showing the feasibility of our approach with real-world simulation models.

The remainder of this paper is structured as follows. In Section 2 we discuss related work. Section 3 introduces our reference PDES system organization. The facilities offered by the ECS distributed middleware are presented in Section 4. Experimental results showing the viability of our solution are provided in Section 5.

2 RELATED WORK

In the recent years, a lot of research effort has been spent to enable PDES systems to fruitfully exploit (low-cost) resources from the Cloud (or virtualized environments in general) to run large models. Some works have been targeted at studying the effects of hypervisor configurations on the runtime dynamics of PDES systems [25, 26], particularly on the side of virtual machine (VM) scheduling and cross-VM communication. These studies have targeted both conservative and optimistic PDES, as the basis to determine whether the Cloud can represent a fruitful infrastructure for complex and large scale PDES simulations. The exploitation of distributed resources, such as Cloud (spot) resources, is a central target also for our work. However the main difference between what we propose and the previous literature studies is that the latter are still bound to the traditional PEDS programming model. In particular, the considered PDES platforms adhere to the paradigm in which the model developer is forced to reason about LP data separation and cannot implement rely to in-place cross-LP state access. Rather, we target more innovative programming paradigms, such as ECS. We therefore target an orthogonal goal, which nonetheless is of similar relevance.

Full state partitioning as in traditional PDES—with event handlers only accessing the state of a single LP—is a programming model leading to deployment of PDES systems which have been shown to be capable of exploiting extreme-scale distributed infrastructures and supercomputing-oriented facilities [1]. Such platforms are not the central target of our proposal. However, enabling ECS to run on distributed-memory systems opens the way to exploiting differentiated classes of computing clusters (including higher-end ones) in conjunction with the innovation in the offered programming model—which breaks disjointness in the accesses to the LP states by event handlers.

As for the enrichment of the programming facilities in PDES systems, the literature shows solutions oriented to enabling data sharing across LPs. The approach in [3] discusses how LP state sharing might be emulated by using a separate LP hosting the shared data and acting as a centralized server. There, also the notion of *version records* is introduced, where multi versioning is used to maintain shared data in order to cope with read/write operations occurring at different logical times while avoiding unneeded rollbacks. This is an approach similar to the one proposed in [16], where a theoretical presentation of algorithms to implement a Distributed Shared Memory mechanism is provided, and 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 instead of mapping accesses to message-passing, we support in-place access to LP state buffers. Retrieving actual operating system pages is fully transparent to the application and is demanded to the innovative distributed ECS middleware we present. Also, we do not limit sharing to a particular memory portion (such as the state image of the centralized server), since any memory buffer representing a portion of the whole simulation model state can be accessed. Contextually, we provide the support for application-transparent distributed deploy of the PDES system entailing such sharing facilities, thus not limiting the support for state sharing to shared memory machines. This overcomes the limitation of the original ECS runtime support [18], which was bound to a single shared-memory machine.

In [10], the notion of *state query* is introduced, according to which any LP needing the value of a portion of the state that belongs to a different LP 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 [13] proposes to integrate the support for shared state in terms of global variables, by basing the architecture on [6]. Although this proposal supports in-place read/write operations as we do (i.e., LPs 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 LPs as readers/writers on shared variables. Also, it does not scale to distributed memory clusters of machines—like Cloud based clusters. Our proposal avoids all these limitations, by also allowing the sharing of dynamically-allocated buffers within the LP state, 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 [17], where shared data are allowed to be accessed by concurrent LPs 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 solution 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 LP, while still providing parallelism and synchronization transparency. Further, that proposal is limited to shared memory machines, while our primary focus in this paper is to port the ECS enriched programming model onto distributed memory clusters.

The work in [7] proposes a framework targeted at multi-core machines and based on Time Warp, where so called Extended Logical Processes (Ex-LPs), 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 [13]. One core difference between our proposal and the one in [7] 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. 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 LP that takes control touch any valid memory location within the global simulation state without the need for any particular care, just like it occurs in sequential-style programming and related sequential execution scenarios. Overall, we “transactify" the access to memory chunks across different concurrent LPs without the need to mark data portions subject to transactional management by the programmer. Further, as a second core difference, the work in [7] does not support cross-LP accesses on distributed-memory systems, which is the primary target of our work to enable exploiting clusters from the Cloud.

Finally, our proposal has relations with approaches that bridge shared and distributed memory programming in general contexts. Among them, we mention PGAS (Partitioned Global Address Space) [22]. However, these solutions do not cope with virtual time-based speculative synchronization, thus not enabling the local materialization of remote data versions complying with timestamp-ordered accesses. In other words, our solution is already specialized to speculative PDES, while the others would require additional modules to be designed in order to accomplish the same objective. As for PGAS, another difference stands in that it relies on compiler-based instrumentation to intercept memory accesses, and to detect whether they refer to remote data. On the contrary, we rely on kernel level facilities operating at the granularity of individual operating system pages, which avoids paying the cost of running instrumented software at all the accesses.

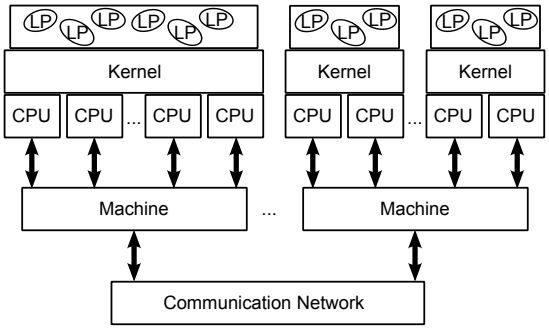

Figure 2: Reference System Organization

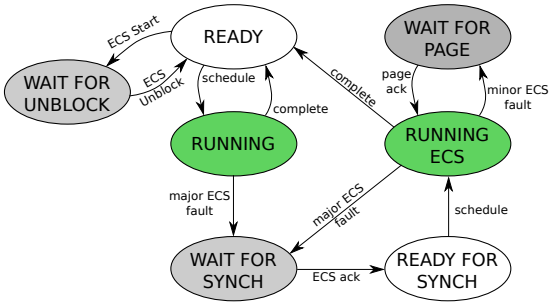

Figure 3: LP State Machine

3 REFERENCE SYSTEM

A high-level schematization of our reference PDES system is depicted in Figure 2. Simulations are supported by a set of (possibly non-homogeneous) processing units, scattered across any number of machines (i.e., computing nodes). On each computing node, any number of *simulation kernel instances* can be running. These instances are developed according to the symmetric multi-threaded paradigm [24], where shared memory is used to support intra-kernel synchronization. Distributed communication is supported by some network interconnection.

According to this organization, a symmetric simulation kernel instance spawns, at simulation startup, a number of concurrent worker threads which is the same as the number of processing units assigned to the kernel instance. Each of these worker threads is stuck to a single processing unit for the whole lifetime of the simulation run. The simulation model's LPs are then assigned to the worker threads according to some *binding rule*. This LP binding ensures that, for a certain interval of wall-clock time, only one worker thread can schedule events destined to one LP. The binding can be recomputed either periodically or depending on runtime parameters, in order to evenly distribute the workload of the simulation on the available computing power.

Therefore, in the most general setting, our reference system model is made up of the following elements:

- A number K of simulation kernel instances (forming up the $KernelSet$), which are scattered across the available computing nodes.
- Each simulation kernel instance $k \in KernelSet$ runs a set of concurrent worker threads, denoted as $TSet_k$. These worker threads rely on shared memory for their internal communication and synchronization tasks.
- At any wall-clock time instant, a worker thread $t \in TSet_k$ is in charge of CPU-dispatching events for a set of bound LPs, denoted as $LPSet_t$. As mentioned before, at any time instant, one LP is managed only by one worker thread. Therefore, $LPSet_i \cap LPSet_j = \emptyset \quad \forall i, j \quad i \neq j$.

Given the distributed nature of the simulation system, at any time an LP_i discriminates between a set of *local LPs*, namely all the LPs bound to any worker thread $w \in TSet_k$ such that $LP_i \in LPSet_t$ and $t, w \in TSet_k$, and a set of *remote LPs*, in any other case.

4 DISTRIBUTED-MEMORY ECS

4.1 Basics

Similarly to the original proposal in [18], our distributed ECS architecture is based on two orthogonal facilities which are transparently offered by the simulation platform. On the one hand, while simulation events are being executed, the platform is able to *detect* that the running LP is accessing the state of another LP, possibly hosted by a remote simulation kernel instance. At the same time, the platform is able to enforce a (distributed) protocol to *synchronize* the Local Clocks of the LPs involved in an ECS synchronization, so as to allow them to observe a consistent view on the simulation state.

In our organization, *cross-state access detection* is provided by innovative kernel-level facilities, which let different worker threads of the platform share the same logical pages although with different access privileges. Therefore, a page fault upon accessing the simulation state of a different LP is the initiation of an ECS synchronization, as it will be later discussed. At the same time, *LP synchronization* is enforced by relying on a (distributed) communication protocol, based on the notion of *control messages*. A control message is a message exchanged across two different LPs, in a way completely similar to event transfer. Nevertheless, with one single exception, control messages are not incorporated into the receiver's event queue, as they are associated with ephemeral state transitions which must not be replayed upon a rollback operation, and must be purely handled at the level of the PDES platform.

Correctness of the whole simulation is guaranteed by two facts: i) the execution of an event by a LP can be *suspended*; ii) every LP is always in an execution state according to the state machine depicted in Figure 3, which allows the PDES platform to correctly interpret the system events and control messages which target every LP.

As for point i) above, we rely on User-Level Threads (ULT), namely CPU contexts which can be saved and restored at any time instant by a worker thread $t \in TSet_k$. In particular, to give control to a LP, the worker thread in charge of it changes its CPU context, allowing the execution of the event to take place in an isolated environment, which has also its own stack. In this way, whenever the simulation platform takes back control, it might determine that the event's execution has to be temporarily suspended, and it deschedules the running LP (i.e., it restores the CPU context related to the worker thread running in *platform mode*). Later, the worker thread can decide to resume the execution of the suspended event,

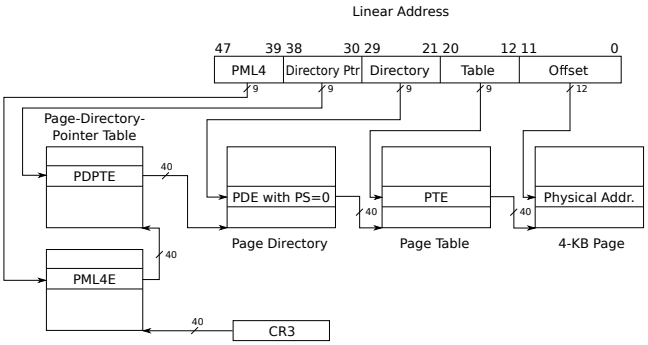

Figure 4: The Paging Scheme in x86_64 Processors.

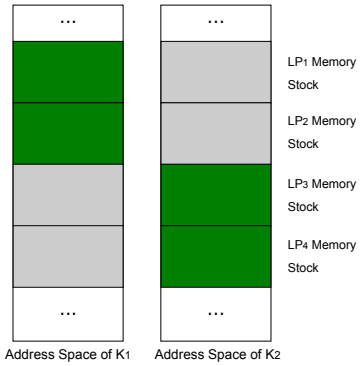

Figure 5: LP Memory Map Organization

and this is done by simply restoring the LP's CPU state. Having a separate stack for every LP within a single worker thread (which has its own system stack) ensures the correctness of the preëmptive event execution. For a thorough technical description of the approach used to realize this facility in an application-transparent manner, we refer the reader to [19].

With respect to point ii) above, the state machine reported in Figure 3 has three different types of states: *blocked states* (gray-shaded) are associated with a LP which has been descheduled while executing an event, thanks to the ULT facility; *ready states* (white-colored) are associated with LPs which can be activated, either to start processing a new event, or to resume the execution of a preëmpted event; *running states*, which are associated with LPs currently executing an event. This organization allows to implement the *smallest-timestamp first* scheduling strategy [12] of each worker thread quite easily, given that only LPs in a ready state can be activated. The transitions across the different states are related to two main kind of events: some are associated with the aforementioned *cross-state access detection*, others with the actual *LP synchronization*. We will thoroughly describe these transitions later.

4.2 Memory Management

In order to support cross-state access detection, the runtime environment must enforce a memory management policy which allows in a simple way to map the memory chunks destined for usage by a LP—via the invocations to the traditional `malloc` library—to a given memory addresses range. This is particularly important given that we must discriminate between memory accesses which target the simulation states of different LPs, which can be either local or remote.

Indeed, the goal is to detect at runtime what LP's state is being targeted by a memory access by relying on pure address-space mapping. When the simulation is started up according to the distributed system model described in Section 3, there are multiple (distributed) processes living in separate virtual address spaces. We therefore need an agreement across the different kernel instances to map LPs states to the same virtual address ranges. Given that we target full transparency towards the application-level programmer, who is allowed to rely as well on dynamic memory allocation, such an agreement could be impossible or over-costly at runtime.

We have therefore resorted, in a way similar to what has been proposed in [21], to a *deterministic memory map manager*. In particular, according to the original shared-memory tailored ECS proposal in [18], each LP is associated with a 1 GB *memory stock*, or a multiple of this memory unit. As illustrated in Figure 5, the base address of this stock is deterministically computed by every simulation kernel instance. In this way, all simulation kernel instances map LP stocks to a same contiguous region of the virtual address space, where the stocks are uniquely associated with an address range which does not overlap.

Given that a simulation-kernel instance manages a pre-defined set of LPs, thanks to its worker threads, at simulation startup these memory stocks are delivered to a fine-grained memory manager, such as the one presented in [20], which ensures that the simulation model's memory requests can be served thanks to traditional APIs, such as `malloc` or `new`.

Overall, this organization delivers memory buffers in a *non-anonymous* way—although transparently with respect to the application—where the buffers destined to serve memory requests by a LP are guaranteed to fall within a memory stock located in a contiguous virtual address region reserved to host the state of that specific LP. In the case of remote LPs, the virtual addresses are initialized and never used to serve memory requests, by all kernels which do not host such LPs (these are the grey regions in Figure 5).

4.3 Kernel-Level Support

Cross-state access detection is ultimately supported by a close interaction with ad-hoc operating system's facilities offered by a custom Loadable Kernel Module (LKM). This module offers two different levels of interaction: *explicit* interaction is supported by a set of `ioctl` commands, to let worker threads notify the kernel when a given LP is starting to process an event; *implicit* interaction allows the kernel to notify the userspace runtime environment whenever a LP is accessing the state of a different LP.

4.3.1 Explicit Interaction. When the module is loaded, it creates the single-access device file `/dev/ecs`. Upon simulation startup, the simulation kernel opens this file to let the module know that its threads must be managed according to the below-described logic, and relies on the `SET_VM_RANGE` `ioctl` command to tell the module what is the range of virtual addresses associated with the LPs.

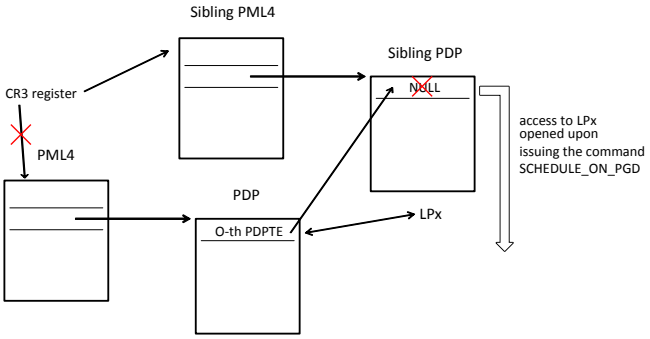

Figure 6: ECS Schedule Example

For the sake of clarity, we report in Figure 4 how a virtual address is mapped to a physical address on x86_64 systems. The CR3 control register keeps a pointer to a first-level paging table. From this table, it is possible to traverse four different levels of indirection, until a physical page is located in memory. The (virtual) linear address is decomposed into five different fields, which determine the offset at each level of the chain where the pointer to the next level is found. The last displacement is the offset within the physical page, into which the memory access is falling.

The memory map depicted in Figure 5 is allocated so that the page table respects an important invariant. We allocate LPs' memory stocks so that the whole GB (or the set of GBs) of memory is aligned to one single entry in the Page Directory Pointer (PDP) table. In this way, any access to any physical page related to the simulation state of a LP can be immediately mapped to the actual LP thanks to the PDP entry used in the virtual-to-physical address resolution. Therefore, thanks to the enforced deterministic memory allocation scheme, the payload of the `SET_VM_RANGE ioctl` command is simply the initial address of the first memory stock reserved for LP_0, and the total number of bytes reserved for the states of all the LPs.

To actually determine when a LP is accessing the state of a different LP, worker threads inform the kernel module what is the LP which will be activated for event execution via another `ioctl` command named `SCHEDULE_ON_PGD`. This command activates a kernel-level logic implemented in the module which installs a *sibling page table* on the CR3 register of the CPU core running the worker thread. In particular, the invocation of the `SCHEDULE_ON_PGD` command puts in place the policy illustrated in Figure 6. The sibling page table is constructed by relying on a cloned PML4 table associated with the virtual memory of the whole process—this can be easily retrieved by the module from `current->mm->pgd`—and by a clone of the PDP tables which point to the simulation state of any LP, be it local or remote. These cloned PDP tables are zeroed in the entries reserved for the LP states, except for the entry associated with the currently-scheduled LP (notified via the `ioctl` call) so that whenever an access is made towards a different LP's simulation state, it generates a memory fault.

Having different sibling PML4 tables associated with the different concurrent worker threads leads to the possibility to concurrently dispatch and execute different LPs—this is done by having each worker thread opening the access to the stocks associated with the

Algorithm 1 ECS Page Fault Kernel Handler

```
1:  procedure FaultHandler(pt_regs* regs)
2:      if current → mm = NULL then                          ▷ F1
3:          DoPageFault( )
4:          return
5:      if current → pid is not registered then              ▷ F2
6:          DoPageFault( )
7:          return
8:      target ← ReadCR2( )
9:      if PML4(target) not in LP range then                 ▷ F3
10:         DoPageFault( )
11:         return
12:     else
13:         if PDP(target) = NULL then                       ▷ F4
14:             fault_type ← Major
15:         else
16:             if GetPteStickyBit(target) then              ▷ F5
17:                 fault_type ← Minor
18:                 SetPresenceBit(target)
19:             else
20:                 if ¬GetPresenceBit(target) then          ▷ F6
21:                     DoPageFault( )
22:                     if GetPdeStickyBit(target) then
23:                         fault_type ← Minor               ▷ F7
24:                         SetPageStickyFlag(target)
25:                     else
26:                         return
27:                 else                                     ▷ F8
28:                     fault_type ← AccessChange
29:                     SetPagePrivilege(target, WRITE)
30:     Switch to the original Page Table                    ▷ F9
31:     Copy to userspace fault information
32:     Push on userspace stack regs → ip
33:     regs → ip ← EcsHandler                               ▷ F10
```

LP it is currently dispatching—while still having the possibility to determine whether any of the dispatched LPs is confining its memory references within its own stocks. The assumption underlying this type of organization is that, when there is the need for opening access to a given stock, the corresponding memory management information is already present in the associated PDP entry of the original page tables. This is not guaranteed by simply validating virtual memory addresses via `mmap`, which leaves memory into the empty-zero state. To overcome this problem, when we initialize the memory map depicted in Figure 5, beyond calling `mmap`, we also explicitly write a null byte into one single virtual page of the stock. In this way, the Linux kernel traps the access to empty-zero memory and allocates the whole chain of page tables for managing the pages within the stock (although a single one of these pages is really allocated). This guarantees the existence of the PDP entry associated with the stock, to be filled into the corresponding sibling PDP entry upon dispatching the LP owning the stock. We note that relying on more traditional facilities, such as `mprotect` would not be viable. Indeed, this would setup policies which are enforced for the whole process, while our approach allows different threads within the same process (the simulation kernel) to observe different memory access privileges, at a negligible cost.

4.3.2 Implicit Interaction. In order to let the userspace runtime environment know when a LP is accessing a different LP's simulation state, we have to intercept the artificial memory faults which are generated by the sibling page table installed in the CR3 register of every CPU core. To this end, when the LKM is loaded, it changes the IDT table (directly accessible via the IDT register) in order to make the pointer to the page-fault handler point to an ad-hoc ECS fault handler (rather than the original do_page_fault function within the Linux kernel) implemented within the module. This ad-hoc ECS fault handler is the core of the detection of a cross-state access, and its pseudocode is reported in Algorithm 1.

Once the ECS fault handler is activated, it first checks whether the handler is activated to resolve a minor page fault (**F1**) or if the fault is associated with the thread of a non-registered process (**F2**), i.e., a process which did not open the /dev/ecs device file. In both cases, it calls the traditional kernel's fault handler and then returns, as the fault has been resolved elsewhere. If the thread is registered with the LKM—it is a thread running within the PDES system—we retrieve from the CR2 control register the *target* address of the memory fault. We first check whether this address belongs to a PML4 entry which keeps LPs memory stocks (**F3**) because, in the negative case, this is a memory fault at the level of the simulation platform which must be resolved via the traditional DoPageFault kernel facility.

We then discriminate what kind of access the LP is making to other LPs. In particular, if the PDP entry associated with the target address is zeroed (**F4**), this means that we are accessing the simulation state of a different LP for the first time. This is the case thanks to the fact that upon scheduling a LP, the SCHEDULE_ON_PGD ioctl command explicitly clears all PDP entries pointing to the memory stocks reserved for different LPs. We refer to this situation as an *ECS Major Fault*. In this case, we give back control to the simulation platform by modifying the instruction pointer's value to make it point to the EcsHandler platform function (**F10**), which will be later described. Before doing this (**F9**), we copy to userspace (in a per-thread buffer) all the information related to the fault (namely the fault type, the faulting memory target, and the address of the faulting instruction), we switch to the original page table by reinstalling into CR3 the original PML4 address found at current->mm->pgd, and we push on userspace stack the original value of the instruction pointer, to let the execution flow be eventually resumed.

The userspace ECS handler, discussed in details in Section 4.5, starts a (distributed) synchronization protocol across the involved LPs, to let them observe a consistent snapshot. When synchronizing towards a remote LP, the LKM has to determine what are the memory pages accessed both in read and write mode, to fetch this content from the remote process hosting the LP. To this end, the userspace handler eventually invokes a LKM facility via the SET_PAGE_PROTECTION ioctl command. The logic associated with this command is similar in spirit to what an invocation of mprotect would do on the stock. As said, we cannot rely on it as it would modify the memory view for all threads.

Conversely, we exploit the organization of a Page-Table Entry (PTE), which is depicted in Figure 7, in the original memory view. In particular, we scan all PTE entries which can be reached starting from the PDE entry associated with the given remote LP towards which the scheduled one is synchronizing. All non-null PTE entries,

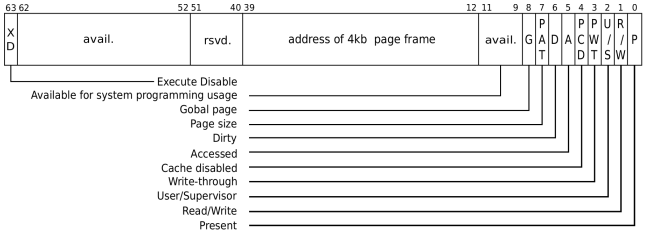

Figure 7: Page-Table Entry (4KB Page)

which are thus associated with an actual materialized page, have the *presence bit* (bit 0) set to 1, to indicate that the Page Base Address is a valid (physical) base pointer for the page. We explicitly force the presence bit to zero, thus generating an additional artificial memory fault whenever such a page (installed by the userspace handler) is accessed. To discriminate whether a fault is artificial or not, due to the above-described scheme, before clearing the presence bit we set bit 9 in the same PTE entry. This is a *programmer's available bit* that we use as a sticky bit—a bit which can be exploited by the LKM to implement additional facilities not supported by the processor firmware. While performing this action, we similarly set one available bit in the PDE entry, to mark the whole memory stock as associated with a remote LP.

Eventually, the LP which initiated the ECS synchronization is re-scheduled, the sibling page table is loaded into the CR3 register of the core where the worker thread is currently running on, and the cleared presence bit will generate a memory fault. This condition is reflected in Algorithm 1 at points **F5**, **F6**, and **F7**. The fault handler first determines whether the page is already materialized, possibly due to a previous execution of an ECS synchronization, by checking if the sticky flag in the associated PTE is set (**F5**). In this case, the presence bit is set back to 1, and an ECS Minor Fault is delivered to the userspace handler, to start the retrieval of the remote pages actually involved in the memory access. Conversely, if the sticky bit is not set, we have to materialize the page if and only if the presence bit in the PTE is not set (**F6**). In this case, we call the original do_page_fault kernel handler. We now discriminate again whether this is a memory fault related to the access to non-materialized pages of local vs remote LPs, by checking the sticky bit in the PDE entry which was previously set. In this case (**F7**), we activate the userspace handler notifying an ECS Minor Fault to retrieve the remote pages, only after having set as well the sticky bit in the PTE entry, to realign the page table to a consistent state according to the logic of the fault handler.

The check at **F6** is important, as it covers as well an additional case. When a LP accesses a remote page in read mode, we explicitly prevent the possibility to access the local copy of the page installed by the userspace handler in write mode by setting bit 1 of the associated PTE to zero. This bit (see Figure 7) is the *read/write bit* which, when set to zero, generates a memory fault when the page is accessed in write mode. In this case (**F8**) we explicitly set back this bit to 1, enabling the possibility to write the page, and deliver an ECS Access Change Fault to the userspace handler. This is an important aspect, as we will later show how this can optimize the finalization of the ECS protocol, in terms of write back actions of

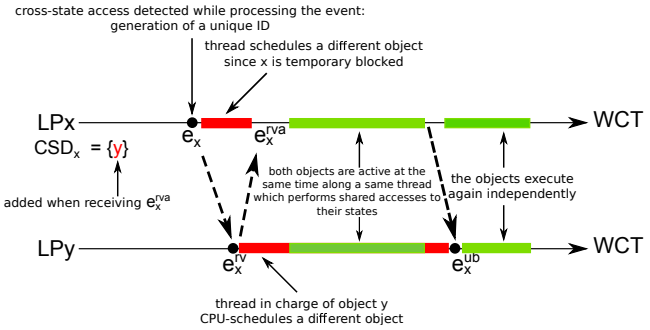

Figure 8: LP Synchronization along Wall-Clock-Time (WCT)

dirty pages towards the node that hosts the master copy of the LP state involved in ECS accesses.

As an additional note, if the target LP involved in ECS is hosted by the the same machine where the source LP resides, the target LP has its operating system pages already locally mapped in the address space. Hence no write-back policy of dirty pages toward the original node needs to be put in place. I this case the sibling page tables setup unleashes full read/write access to the target LP state—based on the `SCHEDULE_ON_PGD` `ioctl` command issue at the PEDS platforms level—thus saving the costs for managing read vs write faults.

Further, our solution is able to handle both 4KB page size (which exactly relies on all the 4 levels of paging we described above) and large pages, namely 2MB pages. In the latter case, the sibling chain that maps a 2MB page will only entail 3 levels of page-tables, namely PML4/PDP/PDE. In fact, our custom fault handler, while traversing the original chain of page-tables, is able to determine whether the target page is a large one or not, and to setup the sibling page-tables' chain accordingly. We exploited `swapoff/swapon` services natively offered by Linux in order to temporarily avoid asynchronous modifications of the original page-tables' chain due to page swapping by the `kswapd` daemon, which would otherwise interfere with our management of the page-table entries.

4.4 The LP Synchronization Protocol

Before entering in the details of the userspace ECS handler, we discuss the (distributed) protocol to synchronize two LPs whenever a cross-state access is detected. Synchronization is supported by control-message passing among the involved nodes. The basic scheme is depicted in Figure 8.

Cross-state accesses must be supported in such a way to ensure that the state snapshot observed by the event-handler is consistent, although generated by a speculative execution. Hence, the LPs whose states are accessed while processing an individual event all need to figure as aligned (in logical time) to the timestamp of the event. This is achieved by encapsulating the cross-state access within an atomic action that is, in its turn, based on an ad-hoc synchronization protocol triggered on demand, if and only if a cross-state access is detected.

The synchronization starts by having LP_x at which the cross-state access is detected send a *rendezvous start* control message e_x^{rv} tagged with a system-wide unique mark[2] towards the destination LP_y. LP_x's execution is then suspended, thanks to the above-mentioned ULT facilities, and it enters the *Wait For Synch* state described in Section 4.1. Once this control message is received and incorporated into the destination LP's event queue, LP_y will eventually reach this event either thanks to forward execution of events in the queue, or due to a rollback operation if e_x^{rv} is a straggler message. The logic associated with the processing of e_x^{rv} is that LP_y is put in the *Wait for Unblock* state and sends back to LP_x a *rendezvous ack* control message e_x^{rva}. Once e_x^{rva} is delivered at LP_x, it moves LP_x to the *Ready for Synch* state, which eventually leads LP_x to be reactivated. The id of LP_y is added to the *Cross-State Dependency* table of LP_x (CSD_x), which is passed as an argument of the `SCHEDULE_ON_PGD` `ioctl` command to determine what PDP entries should be opened for access in the sibling page table temporarily installed in the `CR3` register of the core running the worker thread.

At this point, LP_x and LP_y are aligned to the same logical time instant, and LP_x can access the state of LP_y. In case LP_y is remote, these accesses will generate additional page faults. These will be associated with additional control messages, as discussed later in Section 4.5. This scheme can be iterated multiple times, so that within the execution of a single event, LP_x can synchronize with any number of LPs. The same rendezvous mark is used to track the synchronization, so that in case any of the LPs undergoes a rollback operation, all synchronized LPs can be rolled back as well[3]. The ECS synchronization terminates when LP_x completes the execution of the currently-scheduled event. At this time, it sends to all synchronized LPs a *rendezvous unblock* message e_x^{ub}, so that all LPs can now start again executing independently.

By the above description, the materialization of a cross-state access leads to a non-persistent relation between two or more LPs. In fact, given that cross-state synchronization is operated on a per-event basis, after the finalization of the event that led to cross-state accesses, the involved LPs start again executing alone along their own simulation trajectories. However, in general contexts, a cross-state access by the application code could be the evidence that two (or more) LPs are actually starting to execute in a synergistic way, in terms of overall simulation model execution trajectory.

4.5 Userspace ECS Management

When the LKM notifies the runtime environment that two LPs have to be ECS-synchronized, the handler depicted in Algorithm 2 is activated. This handler performs different actions depending on the type of ECS fault which is notified by the LKM.

The ECS Major Fault case (**H1**) is associated with the initiation of the (distributed) protocol described in Section 4.4. First, a system-wide unique mark is generated, and a rendezvous start message is sent to the LP keeping the portion of the simulation state which is being accessed by the currently-scheduled event. The id of the target LP is delivered by the LKM, as it is uniquely associated with the PDP entry related to the faulting memory address. The running LP then enters the *Wait for Synch* state, and the target LP

[2]These marks are fastly generated by relying on the Cantor Pairing Function using the global id of the LP and a local monotonic counter.

[3]For a thorough description of the rollback strategy and all its implications on liveliness and correctness of the approach, we refer the reader to [20].

Algorithm 2 Userspace ECS Handler

```
1:  procedure EcsHandler(type, info)
2:      if type = Major then                                   ▷ H1
3:          ECS_mark ← generate_mark( )
4:          Send(RENDEZVOUS, info.targetLP, currentLVT)
5:          LP_state ← WAIT_FOR_SYNCH
6:          CSD ← CSD ∪ {info.targetLP}
7:          Deschedule( )
8:      else if type = Minor then                              ▷ H2
9:          disasm ← Disassemble(info.rip)
10:         write_mode ← disasm.write
11:         page_addr ← BaseAddr(info.target)
12:         pages ← PgCount(info.target, disasm.span)
13:         if write_mode then
14:             AddToWriteList(page_addr, pages)
15:         else
16:             AddToReadList(page_addr, pages)
17:         Send(PAGE_LEASE, info.targetLP, currentLVT)
18:         LP_state ← WAIT_FOR_PAGE
19:         Deschedule( )
20:     else if type = AccessChange then                       ▷ H3
21:         page_addr ← BaseAddr(info.target)
22:         AddToWriteList(page_addr, 1)
```

is added to the CSD of the running LP. Finally, the running LP is descheduled thanks to the ULT facilities described before. In this way, the running LP will never be activated until the rendezvous ack is received, as previously described.

The ECS Minor Fault case (**H2**), which is associated only with the access to the simulation state of a remote LP, has to first identify what kind of operation is being executed on the shared state, namely a read or a write operation. This information is only kept in the low-level assembly instruction which has triggered the ECS synchronization. Therefore, we rely on in-place dynamic disassembly of such an instruction, which can be immediately found in the model's address space by looking at the address which caused the memory fault. Again, this information is delivered to the userspace handler by the LKM, together with a snapshot of the relevant CPU registers as observed by the faulting instruction.

The disassembler[4] provides several relevant pieces of information regarding the faulting instruction. Among these, we can determine whether the instruction is accessing in read or write mode, and the size of the memory access. The latter information is used in conjunction with the target memory address where the instruction has faulted, as notified by the LKM, to determine the base address of the first (remote) page which has to be transferred to the local node, and the number of pages. This information is sent to the destination LP as an additional control message, named `PAGE_LEASE`, before putting the LP in the blocked *Wait for Page* state. Once this control message is received at the target LP. Since the target LP is already in a blocked state, we are actually acquiring a *lease* on the pages, having the LP which originated the ECS synchronization keep a temporary master copy of the content of that portion of the state. These pages can be safely installed into the local address space,

[4]In our implementation, we have used the x86 disassembler provided by hijacker, which is available at https://github.com/HPDCS/hijacker.

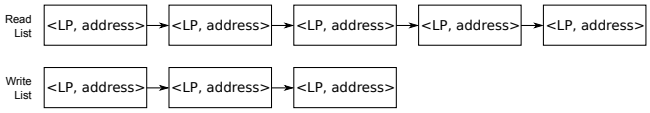

Figure 9: Page Touch Lists

thanks to the non-overlapping organization of the memory map manager depicted in Figure 5.

To keep track of what pages have been leased by a LP, we maintain two *page-touch lists*. One list is associated with pages accessed in read mode, which we refer to as *read list*, while the other (the *write list*) is associated with pages accessed in write mode. The two lists keep as well the id of the LP the original page belongs to, as depicted in Figure 9. This is due to the fact that during the execution of an event, a LP is allowed to synchronize with any number of LPs, thus we must keep track of the ownership of each page. To reduce the complexity of this management, each node in the list is associated with a PTE entry, where a bitmap of 512 bits (one for each page) is used to determine whether the corresponding page has been locally acquired or not.

When accessing a page in read mode, the LP has already acquired a lease on it and a copy of the page content is already installed in the local address space. Since the underlying operating system has granted access in read mode only, once the event handler accesses the same page in write mode, a new fault is detected. Nevertheless, this latter fault can be resolved locally. When activating the userspace handler for an ECS Access Change Fault (**H3**), the LKM has already upgraded the access privilege to write mode. The userspace has only to move the page from the read list to the write list, in the corresponding PTE node.

The two lists are used as well upon the finalization of an event involved in an ECS synchronization. Once the event's execution is completed, the runtime environment has to send a rendezvous unblock control message, in order to notify the synchronized LPs that they can resume their normal execution. The semantic of this event is augmented by adding to it a payload which is composed of all the pages for which a lease in write mode has been acquired during the execution of the event. This allows the destination kernel instances to update the content of the simulation state of the involved LPs according to a write-back scheme, just before giving back control to them. In this way, the states are reconciliated, and every LP in the system can observe a simulation state snapshot which is consistent with the logic of the event handler just executed at the ECS originating LP.

4.6 Memory Reclaim

Due to our organization, the amount of pages materialized on a local node for remote LPs is always increasing. We have devised a simple memory reclaim policy which entails to periodically reset the memory map organization described in Section 4.2.

This operation is supported by having one single worker thread at each node invoke a sequence of `munmap`/`mmap` for every memory stock associated to remote LPs. In this way, we instruct the underlying operating system to release all memory pages which have been materialized during the execution of remote cross-state

synchronizations. It is fundamental to execute this operation in isolation, i.e. when no other thread has any operation related to a remote ECS still pending. In our implementation, we have resorted to a periodic check, with a period of around 30 seconds, where all threads notify through shared variables whether they have pending remote synchronization, and an additional shared variable is used to delay the initiation of a remote ECS if a memory reclaim phase is in progress.

5 EXPERIMENTAL ASSESSMENT

5.1 Testbed Platform

Our ECS distributed middleware has been integrated in the ROOT-SIM open source PDES platform [23], which is used as the testbed PDES system in our experimental study. This platform offers to the users the possibility to run on a fully shared memory machine, or on a cluster of distributed memory machines. In the latter case communication between remote nodes exploits MPI. This same layer has been exploited in the ECS distributed middleware in order to implement both the message exchange protocol that makes threads running on different nodes coordinate with each other, and the actual transfer of virtual pages associated with the LPs' state from one node to another.

We have run experiments on a cluster of virtualized nodes, composed of Virtual Machines equipped with AMD Opteron 2.6 GHz vCPUs, that have been deployed on a private Cloud infrastructure. The VMs are hosted by the VMware Workstation hypervisor (version 10.0.4 build-2249910) hosted on top of a HP ProLiant server equipped with 100 GB of RAM and 8 AMD Opteron 6376 CPUs running at 2.6 GHz. Each one has four cores (for a total of 32 physical cores). We have installed Debian 6.0.7 with Linux kernel 3.2. Each VM has 8 GB of memory, and we run experiments with single-vCPU and dual-vCPU configurations of the VMs, thus overall mimicking different configurations of a cluster of mid-range computing nodes. All the available vCPUs are used by `ROOT-Sim` to carry out the simulation.

5.2 Benchmark Applications

To assess the viability of our proposal we provide data related to two different simulation models, each of which is presented in this section.

5.2.1 Multi-robot Exploration. As first benchmark application, we use the same multi-robot exploration and mapping simulation model that has been used for the experiments whose outcomes have been presented in Figure 1. This model has been developed according to the results in [11], and is based on a group of robots set out into an unknown space, with the goal of exploring it. The map of the explored space is constructed by relying on the notion of *exploration frontier*. By keeping a representation of the explored world, the robot is able to detect which is the closest unexplored area it can reach, computes the fastest way to reach it and continues the exploration. The overall space to be explored is bi-dimensional, and is partitioned into adjacent hexagonal regions, where obstacles are setup in order to limit the freedom of robot move. Each region is such that the evolution of a specific phenomenon, such as wind or a fire event, is modeled via proper simulation events occurring at the LP modeling the region. This kind of model is useful for mimicking a scenario where an open space is modified by, e.g., an accident and the robots are used to explore it for, e.g., rescue activities.

The robots explore independently of each other until one coincidentally detects another robot in its proximity. Whenever two robots enter a proximity region they verify the goodness of their position hypothesis by creating a rendez-vous point, and trying to meet again there. If the hypothesis is verified, they exchange the data acquired during the exploration, thus reducing the exploration time and allowing for a more accurate decision of the actions to be taken. Additionally, if the robots actually meet in the rendez-vous point, it means that the estimation of their respective position is correct. Therefore, they can form a *cluster* and start exploring the environment in a collaborative way.

In an implementation of this model based on the traditional PDES paradigm—relying on disjointness of the accesses to the LPs' states by the event-handler—the discovery of the presence of a nearby robot requires that LPs modeling the robots communicate to each other their current position, or they have to notify their individual positions to specific LPs (i.e., the regions). In either case, explicit cross-LP exchange of simulation events is requested. The same is true for the exchange of knowledge on the exploration process across robots.

On the other hand, the ECS programming model allows for a completely transparent synchronization of the LPs involved in any mutual state update, which therefore simplifies the development process of the simulation model. In fact, as already hinted in Section 1, ECS-based coding of this model saves up to 25% of C-based code lines in event handlers. More in detail, each LP modeling a region 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 region or not. By relying on a fast bitmap scan, each robot is able to discover which robots are present in the region. This is done by relying on a code block where the modeler is not required to rely on any platform-specific API—rather he simply exploits pointer-based access to whatever region's state when processing an event at a robot, namely the region entrance event. The event-handler can access the region bitmap to detect other robots and to indicate it is currently standing into that region. Also, the robot can acquire information about the environment by directly reading this information from the region state, still thanks to ECS support. If one robot is found to already be in the cell, then the newly entering robot simply "merges" its view of the environment. This can be easily done by still relying on direct access to the other robot's state.

In our simulations the model was configured to have 484 LPs modeling individual regions, and their evolution along simulation time, and 4 LPs modeling the robots exploring the overall space.

5.2.2 Data-Grid. As second benchmark application, we rely on a data-grid simulation. It is based on distributed/replicated cache servers, each keeping a subset of the whole set of keys in the entire data-set. Particularly, we consider a model where atomicity of the distributed updates is ensured by running the 2-Phase-Commit (2PC) protocol across the nodes keeping keys that belong to the write set of the transactional operations. In this model, the simulated

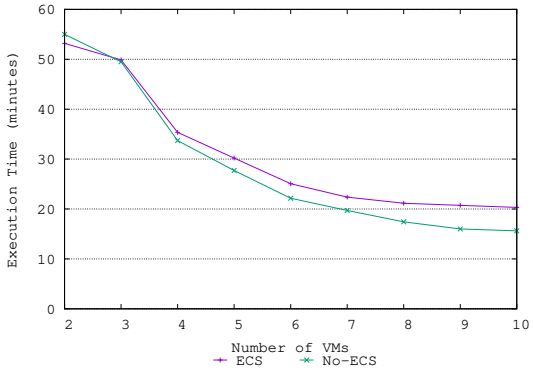

Figure 10: Execution Time - Multi-robot Model

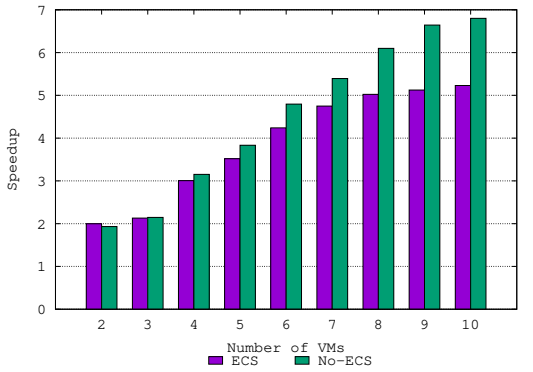

Figure 11: Speedup with respect to Sequential Simulation - Multi-robot Model

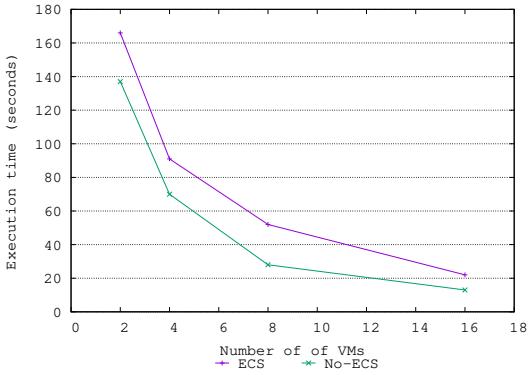

Figure 12: Execution Time Results for the Data-Grid Model with Single-vCPU VMs

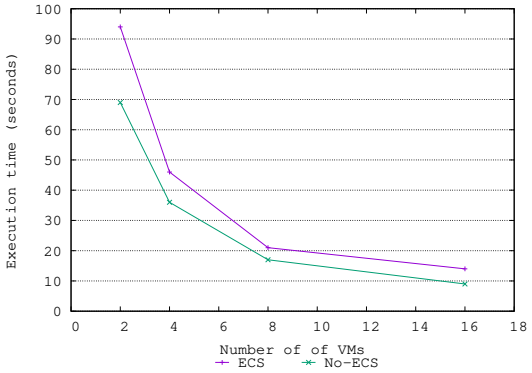

Figure 13: Execution Time Results for the Data-Grid Model with Dual-vCPU VMs

transaction coordinator needs to schedule the arrival of a *prepare request* event to the involved sites—those keeping the replicas of the data it locally handles—which needs to carry information about the write set. These sets may entail hundreds of data-item keys—numerical IDs in our implementation—and are populated at the coordinator while simulating the execution of the transactional task. They are therefore instantiated by the transaction-coordinator LP within its local state. For this model we consider two different implementations, one not relying on ECS, which transmits the write set as the payload of the *prepare request*—for this configuration the programmer is in charge of explicitly coding the pack/unpack of the write set—and another one based on ECS, where the write sets are directly accessed via pointers by the involved simulated nodes (hence the *prepare request* event only needs to carry the pointer indicating where to find the information related to the simulated 2PC phase).

We simulated a data-grid system with 256 nodes (with degree of replication 2 of each $\langle key, value \rangle$ pair in the data-grid), with closed-system configuration in terms of number of clients (and hence number of transactions) running within the system. Particularly, we set the number of active concurrent clients continuously issuing transactions to 256—embedded into the simulation logic of each cache server. This configuration resembles scenarios where the 256 clients operate as front end servers (co-located with the simulated data-platform nodes) with respect to end-client applications. Also, we set the amount of keys touched in write mode by transactional tasks to the order of 1000, while each cache server keeps 100000 items.

5.3 Results

In Figure 10 we report the variation of the execution time (average over 10 runs) for the multi-robot model while varying the number of VMs up to 10—in the single-vCPU configuration. The data show that, with this model, the distributed ECS version has a good scalability up to 7/8 VMs, which tends to diminish for larger VM counts. In fact, up to 8 VMs the execution time with ECS is no more than 15% worse than the one without ECS, and the speedup over the sequential execution, shown in Figure 11, is essentially linear up to 7 VMs. On the other hand, with larger VM counts the non-ECS version scales slightly better, as expected. However, this improved scalability comes at the cost of more code lines, motivated by the need for coding the model in pure data separation across the LPs.

In Figure 12 we show the execution time results for the data-grid model, with variation of the number of VMs up to 16—in single-vCPU configuration. The data are essentially aligned with those

observed with the multi-robot model with the only difference that ECS allows scaling down the execution time also with larger VM counts. In fact, although showing overhead with respect to the non-ECS version, its reduction in the execution time does not flatten. One motivation stands in the fact that the data-grid model has less frequent LP interactions, with respect to the robot-explore model, which occur only upon the simulated 2PC phase involving multiple data accesses by a transactional task—the individual accesses are in fact simulated locally by each individual LP. Further, while in the multi-robot model cross-LP accesses under ECS have a read/write profile—involving page write-back—in the data-grid model they are mostly read accesses that only inspect the write-set involved in the 2PC kept by the LP simulating the transaction coordinator.

In Figure 13 we show the variation of the execution time for the data-grid model when using VMs equipped with dual-VCPS. In this case each simulation kernel instance has two threads and the interaction across LPs based on cross-LP state accesses are sometimes served by an individual machine within the same address space—rather than from a remote one via page transfers. In such a deployment the distance between the ECS and the non-ECS version is reduced, meaning that with our distributed ECS middleware, infrastructure-level investments related to clusters of more powerful VMs pay-off in terms of improved reduction of the execution time. Overall, competitive tradeoff between infrastructure costs and performance can be achieved while still getting the benefits from the more expressive programming model offered by ECS, as compared to the traditional one relying on disjoint LP state accesses. On the other hand, as discussed in the introduction section, moving to significantly more powerful shared-memory machines—possibly hosted in the Cloud—can even lead ECS to provide, together with a more expressive programming support, improvements in the actual performance of the PDES system.

6 CONCLUSIONS

We have presented a middleware-level architecture that allows the expressive ECS programming model for PDES—originally conceived only for shared-memory machines—to be deployed on distributed-memory clusters. The core target of this work is to enable models coded according to ECS to be efficiently ran on top of (low-cost) Cloud resources. Experimental results with two different real-world simulation models, deployed on a cluster of up to 16 VMs hosted on a private Cloud, show the viability of our proposal, and its effectiveness in fruitfully exploiting increasingly powerful virtual resources. Overall, our proposal allows to improve the actual tradeoffs between the achievable runtime performance, the infrastructure-level investments, and the simplification of code development—the latter thanks to an expressive PDES programming support.

REFERENCES

[1] Peter D. Barnes, Christopher D. Carothers, David R. Jefferson, and Justin M. LaPre. 2013. Warp speed: executing time warp on 1,966,080 cores. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation - SIGSIM-PADS '13*. 327. https://doi.org/10.1145/2486092.2486134

[2] Azzedine Boukerche and Sajal K Das. 1997. Dynamic load balancing strategies for conservative parallel simulations. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS)*. 20–28.

[3] David Bruce. 1995. The treatment of state in optimistic systems. *SIGSIM Simul. Dig.* 25, 1 (July 1995), 40–49. https://doi.org/10.1145/214283.214297

[4] Christopher D. Carothers and Richard M Fujimoto. 2000. Efficient execution of Time Warp programs on heterogeneous, NOW platforms. *IEEE Transactions on Parallel and Distributed Systems* 11, 3 (2000), 299–317.

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

[6] K. Mani Chandy and Rivi Sherman. 1989. Space-time and simulation. *Proceedings of the SCS Multiconference on Distributed Simulation* (1989), 53–57.

[7] Li-li Chen, Ya-shuai Lu, Yi-Ping Yao, Shao-liang Peng, and Ling-da Wu. 2011. 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, 1–9. https://doi.org/10.1109/PADS.2011.5936752

[8] Myongsu Choe and Carl Tropper. 2001. Flow control and dynamic load balancing in Time Warp. *Transactions Soc. Comput. Simul. Int.* 18, 1 (2001), 9–23.

[9] Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2017. 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.

[10] Alessandro Fabbri and Lorenzo Donatiello. 1997. SQTW: a mechanism for state-dependent parallel simulation. Description and experimental study. In *Proceedings of the Workshop on Parallel and Distributed Simulation*. 82–89. https://doi.org/10.1109/PADS.1997.594590

[11] Dieter Fox, Jonathan Ko, Kurt Konolige, Benson Limketkai, Dirk Schulz, and Benjamin Stewart. 2006. Distributed Multirobot Exploration and Mapping. *Proc. IEEE* 94, 7 (2006), 1325–1339. https://doi.org/10.1109/JPROC.2006.876927

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

[13] Kaushik Ghosh and Richard M. Fujimoto. 1991. Parallel Discrete Event Simulation Using Space-Time Memory.. In *Proceedings of the International Conference on Parallel Processing*. CRC Press, 201–208.

[14] David W. Glazer and Carl Tropper. 1993. On process migration and load balancing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems* 4, 3 (1993), 318–327. https://doi.org/10.1109/71.210814

[15] Julius Higiro, Meseret Gebre, and Dhananjai M. Rao. 2017. Multi-tier priority queues and 2-tier ladder queue for managing pending events in sequential and optimistic parallel simulations. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS 2017, Singapore, May 24-26, 2017*. 3–14.

[16] Horst Mehl and Stefan Hammes. 1995. How to integrate shared variables in distributed simulation. *SIGSIM Simulation Digest* 25, 2 (1995), 14–41. https://doi.org/10.1145/233498.233499

[17] Alessandro Pellegrini, Sebastiano Peluso, Francesco Quaglia, and Roberto Vitali. 2016. Transparent speculative parallelization of discrete event simulation applications using global variables. *International Journal of Parallel Programming* (apr 2016). https://doi.org/10.1007/s10766-016-0429-2

[18] Alessandro Pellegrini and Francesco Quaglia. 2014. Transparent multi-core speculative parallelization of DES models with event and cross-state dependencies. In *Proceedings of the 2014 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*. ACM Press, 105–116. https://doi.org/10.1145/2601381.2601398

[19] Alessandro Pellegrini and Francesco Quaglia. 2017. A fine-grain time-sharing Time Warp system. *ACM Transactions on Modeling and Computer Simulation* 27, 2 (May 2017). https://doi.org/10.1145/3013528

[20] Alessandro Pellegrini, Roberto Vitali, Francesco Quaglia, Alessandro Pellegrini, and Francesco Quaglia. 2015. Autonomic state management for optimistic simulation platforms. *IEEE Transactions on Parallel and Distributed Systems* 26, 6 (2015), 1560–1569.

[21] Sebastiano Peluso, Diego Didona, and Francesco Quaglia. 2011. Application transparent migration of simulation objects with generic memory layout. In *Proceedings of the 25th Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 169–177. https://doi.org/10.1109/PADS.2011.5936755

[22] Tim Stitt. 2010. An introduction to the partitioned global address space (PGAS) programming model. (2010).

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

[24] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. 2012. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation (PADS)*. IEEE Computer Society, 211–220. https://doi.org/10.1109/PADS.2012.46

[25] Srikanth B. Yoginath and Kalyan S. Perumalla. 2013. Optimized hypervisor scheduler for parallel discrete event simulations on virtual machine platforms. In *6th International ICST Conference on Simulation Tools and Techniques, SimuTools '13, Cannes, France, March 6-8, 2013*. 1–9.

[26] Srikanth B. Yoginath and Kalyan S. Perumalla. 2015. Efficient parallel discrete event simulation on cloud/virtual machine platforms. *ACM Trans. Model. Comput. Simul.* 26, 1 (2015), 5:1–5:26.