A Distributed Shared-Memory Middleware for Speculative Parallel Discrete Event Simulation

MATTEO PRINCIPE, University of Rome "Tor Vergata", Italy
TOMMASO TOCCI, Barcelona Supercomputing Center, Spain
PIERANGELO DI SANZO, Sapienza, University of Rome, Italy
FRANCESCO QUAGLIA, University of Rome "Tor Vergata", Italy
ALESSANDRO PELLEGRINI, Sapienza, University of Rome, Italy

The large diffusion of multi-core machines has pushed the research in the field of Parallel Discrete Event Simulation (PDES) towards new programming paradigms, based on the exploitation of shared memory. On the opposite side, the advent of Cloud computing—and the possibility to group together many (low-cost) virtual machines to form a distributed-memory cluster capable of hosting simulation applications—has raised the need to bridge shared-memory programming and seamless distributed execution. In this article, we present the design of a distributed middleware that transparently allows a PDES application coded for shared memory systems to run on clusters of (Cloud) resources. Our middleware is based on a synchronization protocol called Event & Cross State (ECS) Synchronization. It allows cross-simulation-object access by event handlers, thus representing a powerful tool for the development of various types of PDES applications. We also provide data for an experimental assessment of our middleware architecture, which has been integrated into the open source ROOT-Sim speculative PDES platform.

ACM Reference Format:
Matteo Principe, Tommaso Tocci, Pierangelo Di Sanzo, Francesco Quaglia, and Alessandro Pellegrini. 2020. A Distributed Shared-Memory Middleware for Speculative Parallel Discrete Event Simulation. *ACM Trans. Model. Comput. Simul.* 1, 1, Article 1 (April 2020), 26 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Traditionally, Parallel Discrete Event Simulation (PDES) applications have been based on the explicit partitioning of the entire simulation model into distinct simulation objects (or Logical Processes—LPs) [17] to be CPU-dispatched concurrently. Simulation objects' states are disjoint, and memory accesses during event processing are confined to the state of the simulation object which is executing the event. Interactions across concurrent objects are only supported by exchanging timestamped simulation events via messages.

Within the frame of this programming model, the PDES literature has been focused on improving the runtime behavior of PDES platforms in various directions, ranging from load balancing [3, 5, 10, 19], to the optimization of rollback management in case of speculative processing [24] and

Authors' addresses: Matteo Principe, University of Rome "Tor Vergata", DICII, Rome, Italy, matteo.principe@students.uniroma2.eu; Tommaso Tocci, Barcelona Supercomputing Center, Barcelona, Spain, tommaso.tocci@bsc.es; Pierangelo Di Sanzo, Sapienza, University of Rome, DIAG, Rome, Italy, disanzo@diag.uniroma1.it; Francesco Quaglia, University of Rome "Tor Vergata", DICII, Rome, 00133, Italy, francesco.quaglia@uniroma2.it; Alessandro Pellegrini, Sapienza, University of Rome, DIAG, Rome, Italy, pellegrini@diag.uniroma1.it.

XXXX-XXXX/2020/4-ART1 $15.00
https://doi.org/0000001.0000001

optimistic objects' synchronization [7, 11], and to the effectiveness of platform-level data structures and algorithms [22, 31].

However, the advent of multi-core machines has given rise to new programming approaches for PDES, based on the idea to exploit shared memory to improve expressiveness and flexibility while coding models. This trend is clearly linked to the more general one of preferring shared memory (compared to message passing) because of its ability to support implicit communication across concurrent objects [14].

Along this research path we find solutions enabling the sharing of subsets of simulation-object attributes [9] or global data [30]. These proposals enable the programmer to store data produced/updated by the execution of some event at an object in such a way that these same data can be directly accessed when later (or concurrently) processing another event at another object—with no need for any explicit cross-object data exchange to be coded in the application.

In this context, we have recently presented a highly flexible synchronization protocol for PDES, which we named *Event and Cross-State* (ECS) Synchronization [33]. With ECS, the programmer can write event handlers that can access any memory location belonging to the state of any simulation object via pointers, thus being able to exploit the aforementioned implicit communication paradigm and to reduce the set of API invocations required to code the model. The code example reported below illustrates how an event passed in input to the event handler can carry a pointer (the `reference` field) used to indicate an arbitrary memory buffer belonging to the overall application state. This pointer is used by the event handler to perform in-place accesses to the pointed area.

```
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 the memory location pointed by the reference' field in the event payload
8          ...
9          *p = q ; // writing the content of the memory location pointed by the reference' field in the event payload
10         ...
11 }
```

Accesses to arbitrary memory locations are supported in both read and write mode, thus providing a very powerful way to implement the event logic (no specific API or message passing service is required to make the state of various simulation objects evolve along simulation time). Any event being processed has the possibility to observe the current state of the overall model by simply reading memory locations belonging to the portions of state associated with the different objects (a cross-state access)—see line 7. Similarly, the event can update the state of any of the model's parts, as it may occur at line 9. At the same time, a runtime system based on ECS still manages the simulation objects concurrently. In the proposed implementation [33], correctness of read/write operations—namely causal consistency of the operations on the basis of data/timestamp dependencies—is transparently supported via the integration of both operating system and user space facilities, particularly tailored to Linux and x86-64 processors. Also, ECS has been conceived with a runtime support natively offering speculative processing capabilities.

However, similarly to the solutions presented in [9, 30], one limitation of the original ECS runtime support is that it was conceived for pure shared-memory platforms. Therefore, if some scaled up computing power or more memory would be requested in order to run more demanding models under ECS in reasonable time, one should be forced to resort to a single instance of higher-end multi-core platform.

To overcome this limitation, in this article we tackle the following question: "can we enable ECS-based simulations to run on clusters of low-cost resources, like (spot) instances from the Cloud?". Enabling such a kind of deploy would allow programmers to still exploit the enriched programming

facilities supported by ECS when coding the application—namely, full and everywhere state access in read/write mode via pointers—and, on the other hand, it would allow the final users to still run large models without the need for buying/renting a costly shared-memory machine equipped with large CPU-core counts. A distributed-memory cluster of (virtual) machines would in fact suffice.

We respond positively to such a question by providing novel Operating System (OS) and platform level capabilities—which make ECS suitable for integration in a distributed middleware—enabling a seamless execution on top of distributed memory systems. Essentially, we provide a novel memory management support based on new OS-kernel functionalities, which virtualizes a unique address space on top of a distributed memory system. At the same time, the novel middleware facilities transparently track per-thread read/write accesses onto this address space in order to trigger the execution of middleware-level tasks that (re)-materialize the memory pages associated with the state of a simulation object—at the correct simulation time—on the (remote) node where the event performing the access is running. In other words, our memory management system implements a kind of lease-based mechanism where an operating system page—and its content related to a given virtual time instant along the model execution—is granted in use to (and materialized on) a given computing node for a while, depending on what is happening in terms of the model execution trajectory and overall state accesses. Although our solution has been implemented in the context of the Linux operating system and is suited for x86-64 processors, the concepts it relies on are highly general, and can be exploited for developing implementations suited for other operating systems and/or architectures.

It is important to note that our ultimate objective is not to consistently improve performance when running models on the ECS-based middleware, compared to traditional PDES runtimes (which would however require the model coder to include explicit communication through specific APIs). Rather, our aim is to bring shared-memory programming and ECS to distributed settings (e.g., Cloud-based clusters), with direct benefits in terms of simplification of the programmer's job, while still guaranteeing adequate model-execution performance. In particular, we show that our middleware guarantees the same performance scalability trends, and the ability to fruitfully exploit increased counts of (virtual) machines, compared to parallel executions based on traditional PDES-style coding. On the other hand, the performance advantages of ECS-based executions in pure shared-memory platforms—which represent the ECS-native computing infrastructure—has been demonstrated experimentally in [33].

Our novel ECS middleware has been integrated within the ROOT-Sim PDES system [34], and is available for download[1]. In this article we also report experimental results showing the feasibility of our approach with a panorama of different simulation models and deploys.

The remainder of this article is structured as follows. In Section 2 we discuss related work. The architecture and the funtionalities of the ECS distributed middleware are presented in Section 3, while a discussion on the execution timeline generated by ECS is presented in Section 4. Experimental results are provided in Section 5.

2 RELATED WORK

In recent years, effort has been put into enabling PDES applications to fruitfully exploit resources from the Cloud—or more generally virtualization technologies. Some proposals have been targeted at studying the effects of hypervisor configurations on the model-execution dynamics [41]. These studies have considered differentiated synchronization schemes for PDES, namely conservative and optimistic [12, 41, 42], and stand as baseline assessments of whether virtualized and Cloud platforms can be beneficial for complex and large scale PDES applications. The exploitation of

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

distributed/virtualized resources, is a central target also in our work. However, the main difference between what we propose and these literature studies is that the latter are still focused on PDES applications/systems adhering to the traditional programming model, based on data separation across simulation objects. Differently, we target the ECS synchronization protocol, which enables in-place access to the state of any simulation object by any event handler. The design of a novel ECS-based runtime middleware enabling PDES applications to be run on distributed (virtual) machines in the Cloud—rather than being limited to a single shared-memory machine as in [33]—represents an objective fully orthogonal to (and of similar relevance of) the one pursued by these literature works.

As for the enrichment of programming facilities, the literature offers few solutions oriented to enabling some form of data sharing across simulation objects. The proposal in [4] discusses how state sharing can 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 when optimistic synchronization is employed. A somehow similar solution can be found in [28], where a theoretical presentation of algorithms to implement a Distributed Shared Memory mechanism is provided, which are aimed at keeping replicated instances of a variable coherent. One of these algorithms is exactly based on managing variables as multi-version lists, where write operations install new versions and read operations find the most suitable version. These literature proposals are different from what we present in this article, given that they map shared variables' read/write accesses to message passing—namely, event schedule operations—at the application level, while we support in-place access to any (by default sharable) buffer within the simulation object state via pointers. In fact, the retrieval of the actual operating system pages towards which the accesses are finally carried out is fully transparent to the application and is demanded from the novel distributed ECS middleware we present. Furthermore, data sharing in our proposal is not limited to a particular memory slice—such as the state image of the centralized server. In fact, we allow shared accesses to any memory buffer representing a portion of the whole simulation model state. Contextually, we provide the support for distributed deployment of the PDES system entailing such sharing facilities, thus bypassing the limitation of the original ECS runtime support [33], namely its confinement to a single shared-memory machine.

In [15], the notion of *state query* is introduced, according to which 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, which invalidates 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 [18] enriches the PDES programming model by integrating the support for shared state in terms of global variables, basing the architecture on [8]. Although this proposal supports in-place read/write operations as we do, it provides no transparency, since the application-level code must explicitly register a simulation object as a reader/writer on shared variables. Also, it does not scale to distributed memory clusters. Our proposal avoids all these limitations, by also allowing the sharing of dynamically-allocated buffers within the object state, for which pre-declaration of the potential need to access cannot be raised at startup—in fact our solution allows coping with scenarios where the actual accesses depend, in an unpredictable way, on the specific model-execution trajectory.

In [30], a programming approach and its runtime support are presented, where shared data in PDES applications are allowed to be accessed by concurrent event handlers without the need to

pre-declare the intention to access them, e.g. via code annotations. 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 target data—depending on the timestamp of the event being processes—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. Further, the proposal in [30] is limited to shared memory machines, while our primary focus in this article is to port ECS to distributed memory clusters.

The work in [9] proposes a framework targeted at multi-core machines and based on Time Warp synchronization [25], 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 implementation of the TM layer is based on [18]. A first core difference between our proposal and the one in [9] is that the latter requires a-priori knowledge of the attributes to be shared, which need to be mapped to TM-managed memory locations manually. Rather, our proposal allows for sharing any memory area from the heap, without the need for a-priori knowledge of whether some sharing can (or will) occur along model-execution; this increases the level of transparency. 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. Further, as a second core difference, the work in [9] does not support cross-object accesses on distributed memory systems, which is instead the primary target of our work—according to the aim of enabling the exploitation of clusters of resources.

Similar considerations can be made for PDES-MAS [26]. It is based on sharing data across simulation objects—specifically, objects modeling agents—by relying on given objects hosting portions of the shared data. This is different from what our ECS middleware enables since in PDES-MAS in-place accesses are not supported. On the other hand, PDES-MAS has ortogonal objectives with respect to our work, such as enabling scalable data sharing via optimized data partitioning/distribution [40] and explicit query mechanisms, like range queries [39].

Still in the context of agent-based modeling and simulation, RepastHPC [20] shares a few low-level architectural concepts with our ECS middleware, since it includes the support for dynamic data (agent) migration. However, in RepastHPC the objective of migration is essentially to increase locality in the access to data (by agents), while our ECS middleware has the objective to transparently allow in-place access to whatever data belonging to the simulation model state on distributed memory platforms, while still enabling speculative processing of simulation events.

Our proposal has also relations with approaches that bridge shared and distributed memory programming in general contexts. Among them, we can mention Partitioned Global Address Space (PGAS) [38], distributed shared memory systems [23], distributed file systems [2] and distributed transactional memory systems [21, 37]. 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 fact these approaches are not able to directly support timestamp-based causality relations across concurrent tasks touching data in read/write mode. In other words, our solution is already specialized for speculative PDES, while others that bridge shared and distributed memory programming would require the development of additional modules and software layers in order to accomplish the same objective, possibly relying on the approach presented in this article.

A looser relation exists between our work and Massive Multi-Player On-line Gaming (MMPOG) architectures, particularly the ones based on data sharing via proxies [29]. Specifically, in our ECS middleware we enable migrating the state of a simulation object across multiple computing nodes, depending on the cross-state accesses that occur. However, our proposal is not limited to migrating predetermined slices of information, as instead it occurs in the MMPOG proxy-based approach.

We finally note that full state partitioning as in traditional PDES—with the event handler only accessing the state of a single target simulation object—is a programming model whose runtime enviroments have been shown to be capable of exploiting extreme scale distributed infrastructures and super-computing oriented facilities [1, 6]. This is a reflection of the fact that partitioning can lead to increased locality and to the avoidance of bottlenecks related to the employment of a single centralized memory system. Such platforms are not the central target of our proposal. However, enabling ECS-style coded models to be run on distributed memory systems allows to exploit differentiated classes of computing clusters (including higher-end and bare-metal ones) in conjunction with the flexibility in the exposed programming model, which breaks disjointness in the accesses to the object states by the event handler. In any case, as we show in our experimental study, the ECS middleware allows performance scalability trends (vs the number of used machines) well matching those provided by traditional PDES runtimes. This is a core aspect enabling the effective exploitation of ECS on generic platforms. At the same time, an improvement of the data access locality when running with the ECS middleware could be achieved by adopting dynamic LPs' clustering schemes like, e.g., the one in [27].

3 THE ECS DISTRIBUTED MIDDLEWARE

3.1 ECS Basics

As hinted, ECS allows an event-handler to read and/or update the state of any simulation object when processing some event at a given target object. The original version of ECS [33] has been conceived for running PDES applications on top of an individual shared-memory machine, and the main technical issues that are addressed by such version of the ECS runtime environment are the following ones:

- providing a memory allocation mechanism to respond to dynamic memory requests by a single simulation object such that the buffers delivered to the different objects are guaranteed to be located on different memory segments—different sets of virtual pages—within the address space;
- providing a mechanism for differentiating, in an application transparent manner, the access rules by the different threads operating in the PDES system to the different memory segments. This is essential to determine whether the processing of an event at a given object by some thread leads to overcoming the boundaries of the memory segment destined to that simulation object;
- providing an application-transparent mechanism for synchronizing threads when the above overcoming of the boundaries of each single object state really occurs. In such a scenario an individual simulation object state might need to be accessed in parallel by more that one thread, which clearly leads to data consistency issues;
- providing a mechanism for materializing the correct memory content when the access to the state of an arbitrary set of objects is performed by a thread. Such content must correspond to the correct state value of the targeted objects at the simulation time of the event being processed. This is the value that would have been observed when reaching that simulation-time point along a sequential execution history of the events.

To reach all the above objectives, the original ECS runtime was based on a novel memory management mechanism—with its implementation tailored to the Linux kernel.

Although all the threads operating within the PDES system live in the same address space, with such a novel memory management mechanism each thread has its own view of memory access permissions on a same range of virtual addresses—a feature that is not supported by conventional operating system memory management services. In particular, each thread has different memory access permissions to the operating systems pages that host the state of the different objects. In this way, a simulation object can be made accessible to a given thread, while being not accessible by another thread. Therefore, if the latter thread attempts an access to the object state because of a cross-state event, then the access can be intercepted at the operating system level, and leads to proper actions at the level of the PDES platform. We note that differentiated access rules to a given memory page is a common operating system practice when handling cross-process (e.g. cross virtual machine) memory sharing, like in memory consolidation schemes. However, it is not supported by conventional operating systems when dealing with page sharing across threads of a same process—such as the active threads living within the multi-threaded PDES application.

The design has been realized for x86-64 processors—although its underlying concepts can be adapted to any other type of processors supporting multi-level paging—and such differentiation has been achieved by replicating the first two levels of the page tables that drive the mapping of logical to physical memory. An example is shown in Figure 1 for the case of two threads. On x86-64 machines, page tables are organized in 4 levels[2], and Figure 1 shows how the first two levels—called PML4 and PDP—are duplicated and setup with different permissions to reach and access given logical memory ranges mapped to physical memory passing through the lower level page tables. The latter in their turn are shared across threads, thus ultimately indicating how the address space of the whole PDES application is located into physical memory. In the depicted example, thread T currently has access to the memory pages belonging to the memory segment destined to host the state of simulation object B, while thread T' has a PDP table configuration that does not allow to reach such pages. While processing an event e destined to object A, as soon as thread T' attempts to access the state of object B, the broken path on the page-table chain reveals a memory fault that is handled by an augmented page-fault manager. This manager brings control back to the user space PDES runtime environment which reserves the access to object B for the faulting thread and enables the object image to reach the correct point in simulation time associated with the event e. If such time is in the future, then the PDES runtime environment suspends the processing of e until the correct alignment is reached. If the simulation time of e is in the past of object B, a rollback is transparently carried out reinstalling the correct snapshot of object B to be accessed in read and/or write mode by the event e.

The temporary suspension of event e's processing is based on the concept of User-Level Threads (ULTs), namely a form of coroutines. Indeed, in the original implementation of the ECS runtime enviroment, each simulation object has its own CPU context, which can be saved and restored at any time instant by a thread. In particular, in order to process a given event e destined to some simulation object, the worker thread in charge of it sets up the CPU context of the target simulation object, allowing the execution of the event to take place in an isolated environment. In this way, whenever the PDES runtime environment takes back control—as an example because of a memory fault caused by the access to the state of another object—it suspends the execution of e by simply moving to another CPU context. Later, the thread can decide to resume the execution of the suspended event, and this is done by simply restoring the CPU context that was previously switched off the CPU. Relying on ULTs is a mandatory choice in our architecture for a twofold

[2]We specifically refer in the discussion to the IA32e paging scheme proper of x86-64 architectures.

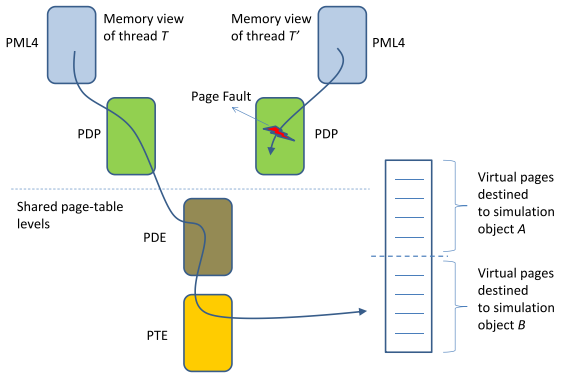

Fig. 1. Differentiated memory views.

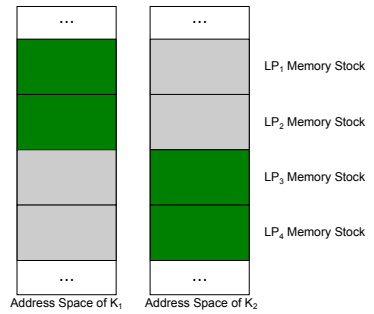

Fig. 2. Distributed memory map organization

reason. On the one hand, once control is given back to the simulation kernel to execute events possibly destined to other LPs[3]—this is the expected behavior for increased throughput while the ECS protocol is synchronizing multiple LPs for (remote) data access—ULTs ensure that separated stacks will prevent functions calls related to parallel execution paths to nest onto each other. If this behavior were not enforced, it would be technically impossible to correctly resume a suspended event. At the same time, since an event could be suspended at any point in its code execution flow at runtime (possibly at each memory access), the compiler cannot ensure that calling conventions to the ECS ad-hoc routines are correctly honored. This latter point might give rise to multiple problems in the overall simulation execution, such as incorrect results, undefined behavior or even crashes. To solve this problem, we rely on a form of ULT which is calling-convention agnostic, and therefore takes a full CPU snapshot. For a thorough technical description of the approach used to realize this facility in an application transparent manner, we refer the reader to [32].

Clearly, when a thread picks an event for processing from the pending event queues of the objects, the destination object memory segment gets opened to the thread for the access to the corresponding pages. This is done via a system call that installs onto the memory view of the thread a path leading to full access to those pages. As an example, in the picture in Figure 1, thread T (the one associated with the left memory view) might have picked an event destined to object B, thus setting up a correct page-table traversal path for reaching the state of this object in memory. While processing this event, additional paths towards other simulation object states can be opened upon the detection of cross-state accesses via the aforementioned memory faults. As soon as a thread finishes processing an event, all the memory paths that were opened are closed, which leads to rejuvenating the overall management for any subsequent event processed by that thread.

3.2 Moving to Distributed Settings

The basic idea for a distributed ECS middleware consists in associating the segment of memory pages related to the state of a simulation object that has been transparently deployed on a remote (virtual) machine to dummy local pages, with no actual meaningful content—these local pages essentially represent buffers, and are dynamically exploited to host the actual copy of the corresponding remote pages when a locally-processed event needs to access the state of the remotely-hosted simulation object.

[3]In the description of our approach, we use LPs and simulation objects interchangeably.

Contextually, all the simulation-kernel instances hosted by the (virtual) machines in the cluster must agree on mapping object states to the same virtual address range, so that using a pointer to a given memory location while processing an event on any machine cannot lead to ambiguous interpretation of what object state should be the target of the memory access operation based on the pointer value. To this end, we resorted, in a way similar to what has been proposed in [36], to a *deterministic memory map manager*, where the stock of memory pages destined to host the state of a simulation object with a given ID are univocally identified via the ID value. An example of combining deterministic memory mapping across multiple machines with dummy pages for representing the state of simulation objects hosted by remote machines is shown in Figure 2. In the example, the managed simulation model has 4 simulation objects, of which the first two (LP_1 and LP_2) are hosted within the address space of simulation kernel K_1 on one machine, while the other two (LP_3 and LP_4) are hosted within the address space of simulation kernel K_2 on another machine. The gray memory segments indicate that a simulation object is not hosted by a kernel instance, but the positioning of the segment—made up of dummy pages—exactly corresponds to the positioning of the actual segment of memory pages reserved for the simulation object on the remote machine where it resides.

Deterministic mapping can be transparently handled by simply redirecting at compile/link time the memory-allocation functions exposed to the simulation model programmer—and used to dynamically instantiate buffers within the object state—to the ECS allocator that delivers buffers located in the proper OS pages. In our implementation, which is bound to the C programming language, these functions correspond to the ones offered by the malloc libray API.

Clearly, when an event e is CPU-dispatched on a given machine, the memory view of the thread processing e is setup so that all the remotely hosted simulation objects—namely the corresponding dummy pages—are not allowed to be accessed, as well as all the locally hosted objects that are different from the object targeted by e. While processing e, if a memory fault is detected, which leads a thread to access the state of a remotely hosted object—the fault occurs on some gray zone of memory—the page-table traversal path would be setup—just like in the original ECS approach—and the actual memory pages that host the target object on the remote machine—better to say, the shapshot of these pages observable at the simulation time of the event e—are retrieved and installed locally onto the corresponding dummy pages. On the other hand, if the fault involves a locally hosted object, no cross-machine page transfer needs to be actuated—in this case we essentially fall in the shared-memory access scenario already tackled by the original version of the ECS runtime support [33].

Noteworthy, this approach needs to carefully consider the tradeoff between (A) the granularity according to which memory accesses and the corresponding read/write access mode are tracked, and (B) the cost for both transferring the correct shapshot of the remotely hosted pages to the local node and sending these pages back to the remote node if the event e generating the cross-state access leads to modifications of the remote object state.

More in detail, the page-table traversal path that allows intercepting whether an event e being executed by a thread is accessing the state of one or another object is set to be defined by the value of some page-table entry at the PDP level—if the entry is not valid (as for the case of thread T' in Figure 1), any memory access to the object state issued while processing e is intercepted and triggers ECS synchronization actions. The reliance on the PDP table (rather than lower level page-tables) guarantees a good tradeoff between the admitted size for the state of each individual object, and the overhead for managing the differentiated memory views. In particular, associating a path starting from an entry of the PDP table to a virtual address range destined to host the state of an individual object allows up to 1GB of virtual memory for that object—this is because PDE end PTE tables have 256 entry each, and an individual page on x86-64 machines is 4KB large. This

default configuration already enables managing large-object models[4]. It also contributes to keep low the overhead for managing the per-thread memory view since it only requires the management of one PML4 and one PDP (see again Figure 1). However, filling dummy pages with up to 1GB of memory by fetching the corresponding set of pages from the remote node hosting the object hit by the event e would lead to unaffordable overhead, especially in contexts where the event e causing the fault and triggering ECS synchronization would access a few bytes into the target object state.

Also, once a page-table path is setup to access the state of some target object when the cross-state event e is being processed, and once the remote pages are transferred onto local dummy pages, we have no control on whether the event handler accessing the object state is writing to the state. In fact, relying on conventional OS memory management, the transfer of a remote page onto a local dummy page leads to implicitly "dirtying" the dummy page—on x86-64 processors this sets the `DIRTY` bit of the corresponding entry of the lowest level PTE page table. This, in turn, leads to the impossibility to discriminate whether event-specific writes actually occur after the fill of the dummy page. In such a scenario, we might need to bring the whole object state back to the source node as soon as the cross-state event completes, independently of what pages in the local state copy were actually written by the event-handler.

We note that this problem cannot be solved by relying on conventional OS memory management services like mprotect, which changes the access permission rule to pages in a memory zone at runtime. In fact, using mprotect to disable write access to the dummy pages after they have been filled with the corresponding remote pages to determine if writes would occur while processing the event e via page-fault handling would lead the OS to reset the memory views of the different threads to the same original view (the one based on the original chain of PML4 and PDP tables on x86-64 processors). This is done to realign any TLB (Translation Lookaside Buffer) storing mapping data for the address space of the simulation application to the newly posted page-access permission values. Overall, such an approach is not compatible with the objective of the ECS runtime system.

The above points raise the need for additional unconventional memory management mechanisms able to detect and properly handle:

- the locality of memory accesses to the pages hosting the state of a remote object and
- the access mode (read vs write) to individual pages of the remote object.

These aspects are intrinsically related to our redesign of the ECS runtime system as a distributed middleware. In fact they were fully irrelevant in the original ECS support purely tailored to shared-memory multi-core machines [33].

To cope with the above two issues, while still relying on per-thread PML4 and PDP page tables, we operate on the lower-level page tables—the ones shared on all the memory views of the different threads (see again Figure 1). In more detail, in the OS-kernel page-fault handler supporting the distributed ECS runtime system, we work on the `PRESENT` bit of each individual page—kept in the PTE level page-table entries on x86-64 processors—in combination with kernel programmers' available bits to put in place a state machine that discriminates whether:

a) the dummy page is already mapped to physical memory, allowing to buffer the content coming from the remote node;
b) the dummy page already hosts the correct copy of the remote page associated with the object state targeted by the event handler;
c) the dummy page has been accessed in read/write mode by the event handler after it was filled by the ECS middleware with the copy of the actual page associated with the state of the target object.

[4]For models requiring more than 1GB for the state of individual objets, we can use more than one entry of the PDP table to map the corresponding virtual pages in memory.

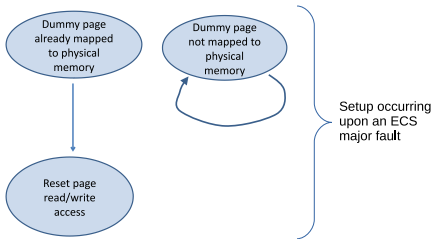

Fig. 3. OS-kernel level page access right manipulation upon an ECS major fault.

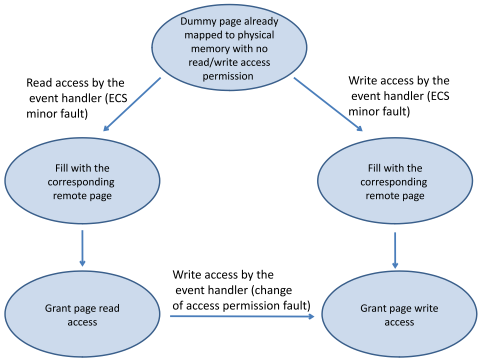

Fig. 4. OS-kernel level evolution of a physical-memory mapped dummy page within the ECS distributed middleware.

We note that the condition in point b) above is not conventional for a page belonging to the address space of any process. Referring to Linux, its kernel never discriminates whether a given page that is mapped to physical memory—for which we know whether some write access has been performed—hosts a given content or not. The same is true for the condition in point c) since, as hinted, the DIRTY bit in the PTE entries of x86-64 machines is a sticky flag, which is unable to track the occurrence of subsequent write operations on a same page—in our case the middleware-level write filling the dummy page with the proper content, and then the write carried out by the event handler (if any) which leads to really dirtying the state of the remote object.

The new page-level state machine initialization is shown in Figure 3. This is put in place by our memory manager operating at the OS-kernel level when the access to the memory segment associated with the state of the target object is granted to the thread running the cross-state event e. As shown, a dummy page that is not yet mapped to physical memory—it has never been used as a buffer for hosting the corresponding page of the object state coming from the remote node—it is simply left as "un-materialized". This allows saving physical memory when, e.g., locality of the accesses to the pages of the state of the remote object never let the event handler fall on that specific page. On the other hand, if the dummy page has been already materialized (it exists in physical memory) its read/write permission is reset. This requires combining the PRESENT bit value with the additional programmers' available bits. This initialization of the state of the dummy pages associated with a remote object takes place upon the first access by the cross-state event e to the memory area associated with that object in the deterministic memory mapping. In such a case, the target remote object, e.g. LP_j, and the one that is currently being executed by the thread as the destination of the event e, e.g. LP_i, need to start a synchronization phase. For this reason we say that the memory fault generated by such a cross state access is an *ECS major fault*, which triggers the execution of a handshaking protocol among local and remote threads.

This synchronization phase leads to retrieving the remote page at its correct snapshot with respect to the timestamp of the event e accessing it, which is then installed onto the corresponding dummy page. Therefore, this dummy page will eventually be mapped to physical memory, with access permission corresponding to the type of access (read vs write) issued by e. As hinted, LP_i simply suspends its execution while the remote page belonging to the state of LP_j is retrieved, and the thread originally running e is allowed to CPU-schedule events destined to other objects. At the same time, once the remote object LP_j has been aligned in simulation time to the timestamp of the

event e, it is suspended (still thanks to ULTs) up to the completion of the processing of e, since its state must be freezed at that simulation time while e performs its accesses. We note that either the logical time of LP_j reaches the timestamp of the event e while processing forward, or LP_j is rolled back to that logical time if it already ran ahead of the timestamp of e.

When the target page P whose access generated the ECS major fault is locally received, the scenario is such that the memory zone made of dummy pages associated with the remote object LP_j is locally accessible, in terms of traversal of the whole chain of page tables containing information for the memory mapping of that zone. However, no page in that zone except P can be actually accessed without the access being traced. Hence, as soon as the event handler tries to access any page other than P in that zone, our augmented OS-kernel page fault handler traces the access and passes control again to the PDES-platform fault handler that retrieves the page from the remote node—in its correct version aligned with the virtual time of the cross-state event being processed—installs it onto the corresponding dummy page, and updates the corresponding PTE entry to indicate whether the performed access is a read or a write. In this case we say that we experience an *ECS minor fault*, since the target remote object, e.g. LP_j in our example description, is already aligned in simulation time to the timestamp of the cross-state event e whose processing causes the fault. Also, if a dummy page was previously filled with the corresponding remote copy, and was set as accessible in read mode, a write operation by the event handler again leads to a fault. However, this fault is processed locally and simply leads to opening write permission to the page, recording that the page has been dirtied by the cross-state event. In this case, it needs to be brought back towards the node hosting the remote object involved in the cross-state access, e.g. LP_j, as soon as the event e is fully processed. In Figure 4 we show the state machine that characterizes the evolution of any dummy page used to host the remote page associated with the state of the accessed remote object when read/write operations causing ECS minor faults are carried out while processing the cross-state event e, or when a fault causing a variation of the access permission rule is experienced.

Clearly, if the accessed page corresponds to a dummy page not yet mapped to physical memory, then our OS-kernel page fault handler initially passes control to the original Linux page fault handler in order to let it map the page. In Algorithm 1 we show the pseudo-code of the PDES-platform page fault handler, whose execution is triggered by the augmented OS-kernel page fault handler when it detects a major or a minor ECS fault involving the access to the state of a remote object. The type of the fault (major vs minor) is passed in input to the algorithm—which executes differentiated actions based on the fault type—together with a buffer containing fine-grain information on the fault, such as the memory address involved in the access, the identifier of the simulation object that has been hit, and a reference to the memory address of the instruction causing the fault. Initially, this fine grain information is used to determine whether the access is in read or write mode, and to identify the target page, which is then inserted into either a read or a write list of pages associated with the hit remote object.

The ECS major fault case (**H1**) is associated with the initiation of the (distributed) protocol that makes the local thread request a lease on the overall state of the remote simulation object. First, a system-wide unique mark is generated, and a rendezvous start message is sent towards the remote simulation kernel. The message piggybacks information identifying the remote page that has been locally touched—by hitting the corresponding local dummy page—as well as the identifier of the hit remote object, namely $targetLP$—this is known by the deterministic memory mapping rule. Then, the LP that was locally running ($currentLP$) enters the *Wait for Synch* state and is CPU-descheduled exploiting the ULT facilities described before. In this way, this LP will never be re-activated until a rendezvous ack is received, indicating that $targetLP$ has reached the simulation time of the event generating the cross-state access, which corresponds to $currentLP.LVT$. The

Algorithm 1 PDES-platform ECS handler

```
1:  procedure EcsHandler(type, info)
2:      disasm ← Disassemble(info.instruction)
3:      write_mode ← disasm.write
4:      page_id ← PAGE(info.target_address)
5:      if write_mode then
6:          AddToWriteList(info.targetLP, page_id)
7:      else
8:          AddToReadList(info.targetLP, page_id)
9:      if type = Major then                                   ▷ H1
10:         ECS_mark ← generate_mark( )
11:         Send(RENDEZVOUS, info.targetLP, currentLP.LVT, page_id)
12:         currentLP.state ← WAIT_FOR_SYNCH
13:         Deschedule(currentLP)
14:     else if type = Minor then                              ▷ H2
15:         Send(PAGE_REQUEST, info.targetLP, currentLVT, page_id)
16:         currentLP.state ← WAIT_FOR_PAGE
17:         Deschedule(currentLP)
18:     else if type = AccessChange then                       ▷ H3
19:         AddToWriteList(info.targetLP, page_id)
```

rendezvous ack message also carries the copy of the remote page that has been hit. This page is then locally installed on the corresponding dummy page, with access permission (read vs write) related to the type of the access performed by the cross-state event.

The ECS minor fault case (**H2**) simply sends a request for the hit page towards the remote kernel hosting the remote object accessed by the cross-state event. Also in this case, *currentLP* is temporarily suspended, until the arrival of the requested page. On the other hand, a fault related to a write operation on a remote page that was already locally installed with read permission due to a previous read operation simply leads to moving the page to the write list (**H3**). In this case no suspension is required for *currentLP*. The latter two cases (**H2** and **H3**) match the page evolution scheme depicted in Figure 4, which is operated at the OS-kernel level.

We do not explicitly report the fault handling logic related to cross-state accesses targeting the state of locally-hosted objects, since it is a simplification of the one in Algorithm 1, and corresponds to the logic already presented in the pure shared-memory version of the ECS runtime support presented in [33].

3.3 Page prefetching

The scheme we have presented in Section 3.2 to support cross-state events accessing remote objects essentially offers to threads leases on segments of logical memory made up of remote virtual pages—in particular their snapshot is aligned at the cross-state event timestamp. This scheme has the additional feature of locally buffering only those pages that are actually accessed along the lease period. This minimizes the communication overhead across the nodes in the system while enabling the local (pointer based) access to an object that was deployed remotely by the PDES platform depending on setup choices.

The communication overhead reduction is also achieved thanks to the fact that at the end of the lease period (namely when the cross-state event accessing the state of the remotely hosted object is fully processed) we can discriminate which pages need to be sent back to the source node, because they have been updated by the event-handler.

However, the policy where a page is requested only upon tracking an access to it, might not minimize the number of memory faults and suspensions of the local simulation object *currentLP* upon processing a cross-state event leading to remote accesses.

In this section we present an approach where a state machine is used at runtime to determine if prefetching of remote memory pages whose corresponding local dummy pages have been not yet

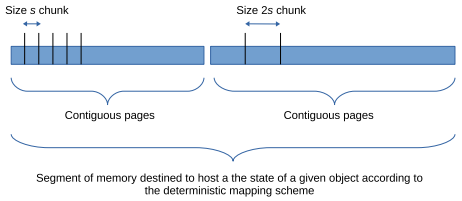

Fig. 5. Allocators with same-size chunks mapped to contiguous memory pages.

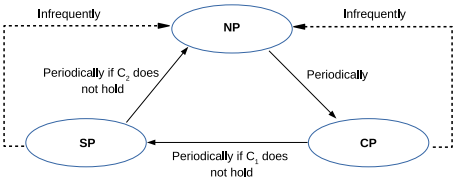

Fig. 6. The state machine driving the selection of prefetch vs non-prefetch policies.

accessed by the cross-state event can be convenient, and according to what prefetching scheme among the two possible alternatives we describe below. Based on the structure of Algorithm 1, if we decide to prefetch multiple pages when an ECS major fault occurs—namely, when sending the rendezvous request to the remote node—then we might have the advantage of reducing the number of ECS minor faults that will occur along the cross-state event timeline. Hence, the state machine driving the activation or deactivation of prefetching at runtime aims at optimizing the tradeoff between the cost of wrong prefetching of pages that will not eventually be accessed by the cross-state event, and the cost of ECS minor faults.

The first prefetching alternative, which we name Clustered Prefetching (CP) can be useful in scenarios where a cross-state event accesses contiguous memory buffers belonging to the state of the remote simulation object. This can be the case of models coded by relying on memory-contiguous data structures such as arrays or large structures, or a combination of them. With CP, the ECS middleware requests N contiguous pages to the remote node hosting the simulation object targeted by the cross-state event upon sending the rendezvous message, with the first of these pages being the one hit by the memory access generating the initial ECS major fault. If $N' < N$ contiguous pages actually host memory chunks delivered for usage to the simulation object, then only N' pages are sent back with the rendezvous acknowledgment.

The second prefetching alternative, which we name Scattered Prefetching (SP), picks the N pages to prefetch randomly. However, randomness has a bias based on the density of memory chunks currently used by the simulation object, which are hosted on zones of contiguous pages. This approach well fits scenarios where the memory allocator exposed to the application-level software installs sequences of chunks of a same size on a given zone of contiguous memory pages, and chunks of another size onto another zone of contiguous memory pages. Then, based on the size of the memory allocation requests coming from the simulation objects, it picks chunks from one or the other zone of contiguous pages. An example of this type of chunk mapping to zones of contiguous pages is schematized in Figure 5, where the left zone is used for hosting chunks of size s and the right zone is used to host chunks of size $2s$. In our implementation we exploit the PDES-oriented DyMeLoR memory allocator [35] which follows this chunk-to-page rule.

A common coding pattern involving a scattered data structure, like a linked list, implemented into the state of a simulation object typically leads to bias the usage of chunks towards a given size—as an example, the size of the record keeping the elements of the list. Therefore we may have higher incidence of usage of the zone of contiguous pages hosting the chunks of that size, compared to other zones of contiguous pages destined to other sizes. In SP each zone of contiguous pages z_i has a weight w_i, and the random selection of the pages to prefetch picks a number of pages in each zone which is proportional to its weight w_i. Overall, given that we would like to prefetch N pages, then the number of pages N_i randomly selected in the zone of contiguous pages z_i hosting chunks of a given size is $N \times w_i$. The weight w_i is computed in our implementation as

the normalized occupancy percentage of each individual zone of contiguous pages hosting chunks of a given size, which depends on how many chunks are allocated from that zone on the average, and on the chunk/zone size. Clearly, if less than $N \times w_i$ pages currently keep allocated chunks in the z_i zone, then only those pages that contain allocated chunks are eligible for prefetching in the random selection. In other words, we do never prefetch pages that keep no buffer currently belonging to the object state layout. Also, the page hit by the memory access causing the ECS major fault is included by default in the set of prefetched pages.

Overall, CP ans SP represent prefetching policies that might fit two different classical scenarios in terms of the actual layout of the state of a simulation object (memory contiguous vs scattered) and of the locality of the accesses an event might carry on into that state.

The state machine that is used to determine whether to prefetch, and in the positive case according to what policy, is depicted in Figure 6. The NP (No-Prefetch) state is such that the runtime system is currently operating without making any prefetch. This is the default initial state of the state machine. After persisting for a given period in the NP state, we switch to CP, and then—after persisting in the CP state for a period of time—we evaluate whether the CP policy is effective compared to NP. This assessment is based on comparing the average number of ECS minor faults per cross-state event that are generated when running with CP, denoted as $minorFaults_{CP}$, and the one that was observed when running with NP, denoted as $minorFaults_{NP}$. In particular, we define a threshold percentage α such that CP can be considered effective if the following inequality holds:

$$minorFaults_{CP} \leq \alpha \times minorFaults_{NP}. \tag{1}$$

If the predicate expressed by Equation (1) does not hold, we switch to SP, and after persisting again in this state for a period of time we compare the average number of ECS minor faults per cross-state event when running under the SP policy, denoted as $minorFaults_{SP}$, with $minorFaults_{NP}$, still using the percentage threshold α. We say that SP is effective with respect to NP if the following inequality holds:

$$minorFaults_{SP} \leq \alpha \times minorFaults_{NP}. \tag{2}$$

If the predicate expressed by Equation (2) does not hold too, then we switch back to NP. In this scenario, the state machine tells that none of the two prefetch policies can effectively cope with the specific access pattern to the object states by cross-state events—the access pattern is implicitly too irregular for being tackled by page prefetching.

Once back in the NP state, we again restart evaluating $minorFaults_{NP}$, and then retry the prefetch policies according to the state machine structure. Such a cyclic retry can be useful in scenarios where the access patterns to the object states by cross-state events can change over time, so that prefetching may become effective at some point along model execution. On the other hand for models with stable access, a single cycle of the state machine could be enough to select the appropriate policy.

A core point that we still need to discuss is how to select the number N of pages to be prefetched with either CP or SP. In our approach, we set N to the average number of remote page accesses by cross-state events when running under NP. Therefore, our idea is to prefetch the same number of pages that we would expect to be actually accessed by a cross-state event—in fact NP fetches only the actually accessed pages. As for α, this is a tunable parameter, for which we suggest values in the interval $[0.1, 0.2]$. Larger values would in fact lead a prefetch policy to be considered as better that NP even if it (potentially) makes excessive errors in prefetching pages that are not really required, while smaller values would lead the state machine to discard a prefetch policy that can instead be effective when compared to NP.

Setting N to the average number of pages of a remote object accessed by a cross-state event is the motivation for having the additional transitions from CP to NP and from SP to NP in the state

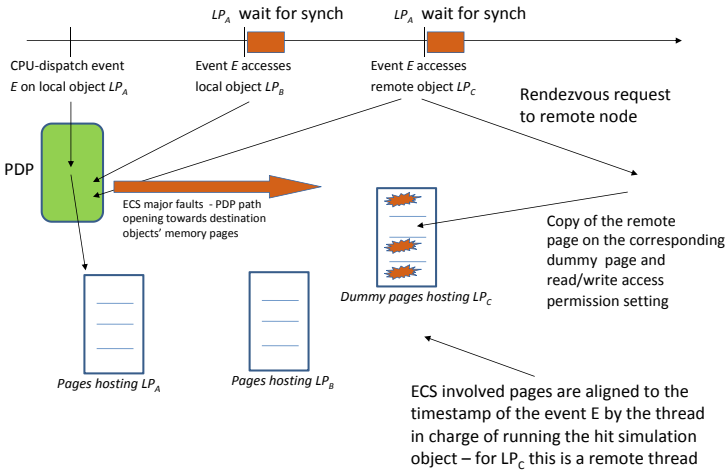

Fig. 7. A timeline with the distributed ECS middleware.

machine. These are labeled as infrequent, meaning that even if we fall into a state that reveals the effectiveness of a prefetch policy on the basis of the value of N estimated before entering that state, we need a mechanism to re-evaluate N along the model execution lifetime, especially for models where the average number of pages of a remote object state accessed by a cross-state event can change over time. Such a variation must in fact lead to a re-evaluation of the actual goodness of the prefetch policies, under the new access pattern, compared to the baseline no-prefetch scheme.

4 THE OVERALL ECS TIMELINE

To provide the reader with a final comprehensive picture of the behavior of the ECS distributed middleware, we present an example execution timeline in Figure 7. Initially, a thread picks the event E from the event queue of the locally hosted object LP_A. Before processing this event, full access to the whole segment of memory pages associated with LP_A is granted to the thread by the ECS support. For simplicity, in the picture such an access is depicted as a path from the PDP page table belonging to the thread view towards the memory pages hosting LP_A. However, we implicitly mean that the path exploits the lower level page tables, namely PDE and PTE, which are shared by the memory views of the different threads. As a consequence, no fault will be ever triggered when accessing the state of LP_A while processing the event E destined to it. We also note that with ECS LP_A can be dispatched if and only if it is not currently involved in any cross-state access generated by an event destined to another object, which is being processed by a concurrent (remote) thread.

Then the event-handler processing E accesses the state of another object, namely LP_B in the example, and an ECS major fault occurs since that object was not originally accessible while executing the event E—it might even have been under the control of another thread that was processing an event bound to it. The ECS fault triggers middleware tasks (a rendezvous request and its acknowledgment, with temporary transition of LP_A to the *Wait for Synch* state) such that the access to LP_B is eventually granted. This occurs after the thread in charge of LP_B aligns LP_B's snapshot in virtual time to the timestamp of the event E. In the meanwhile, LP_A is CPU-descheduled (the red box on the timeline represents the permanence of LP_A into the *Wait for Synch* state). Given

that LP_B is hosted locally by the same machine where the event-handler processing the event E is running, full access to the "master" copy of the state of LP_B is granted, with no need for additional ECS minor faults and the tracking of read/write accesses—these are in fact required for remote page fetch and write-back. At this point both LP_A and LP_B are under the control of the thread executing the event E, and the event-handler can safely proceed performing any access to the states of both.

Then, a third object gets involved in the execution of the event E, namely LP_C. This happens because of an access by the event handler processing E to the memory segment of virtual pages reserved to represent the state of LP_C. In such a scenario, given that LP_C was not originally accessible and it is a remote object, these pages are dummy and need to be filled with the actual content hosted on the remote node—in fact, the master copy of the state of LP_C is remote. In such a case the ECS major fault initially triggered by the access to the state of LP_C leads to opening the path towards the state of LP_C in the faulting-thread memory view, contextually to the closure of any access permission to individual dummy pages hosting the state of LP_C. On the other hand, the page accessed upon that fault gets reclaimed from the remote node (via the rendezvous/ack protocol) and then installed onto the corresponding (local) dummy page—with access permission initially set depending on the type of the issued access[5]. Other pages keeping the master copy of the state of LP_C can be reclaimed on demand if eventually accessed by the event-handler—in fact pages not yet accessed still have no access permission, which leads an actual access to trigger an ECS minor fault passing control to the distributed middleware.

As indicated in the picture, it is the responsibility of the threads managing LP_B and LP_C to bring these objects to the correct simulation time for making the access to the objects states by E causally consistent. As hinted, this may simply require them to reach the timestamp of E in forward execution, or to rollback them if they speculatively ran ahead of the timestamp of E.

As a final note, LP_A might be hit in its turn by a cross-state event E' concurrently processed by some thread and destined to another object LP_D. If the timestamp of E' is in the future of the timestamp of E, then LP_D will be temporarily blocked in a rendezvous up to the point in time when LP_A reaches the timestamp of E'. On the other hand, if the timestamp of E' is in the past of the timestamp of E, LP_A will rollback. If the need for rollback arises while LP_A is in a *Wait for Synch* state, then its execution is no longer resumed in the context of E's processing—in fact E is doomed to be rolled back.

In the online Supplemental Material we describe how the progress of the simulation run can be guaranteed in face of the occurrence of rollbacks while processing events generating cross-state accesses.

5 EXPERIMENTAL STUDY

5.1 Testbed Platform

We have integrated our ECS distributed middleware into the ROOT-Sim open source PDES platform [34], which is used as the testbed PDES system in our experimental study. ROOT-Sim gives 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 is based on MPI3. 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 two different computing platforms. One is a cluster of 16 Virtual Machines (VMs) equipped with AMD Opteron 2.6 GHz vCPUs, which have been deployed on a private Cloud infrastructure. The VMs are hosted by the VMware Workstation hypervisor (version

[5]For simplicity of representation prefetching of additional pages is not included in the picture.

10.0.4 build-2249910) hosted on top of an HP ProLiant server equipped with 100GB 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 3.10. Each VM has 8GB of memory, and we have 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.

The second computing platform is based on a cluster of VMs taken from Amazon Web Services (AWS). In more details, we used 16 VMs, each one being a t2.large VM equipped with vCPUs based on Intel Xeon 2.4GHz processors and 8GB of RAM. Also in this case we used two different configurations of the VMs, namely single vs dual-vCPU. The same software configuration described above has been used for the experiments on the AWS computing platform.

5.2 Experimental Results

5.2.1 Results with a Multi-Robot Exploration Model. As a first benchmark application to assess our distributed ECS middleware, we used a multi-robot exploration and mapping simulation model. Our implementation has been developed according to the model specification provided in [16], 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. 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. Each robot is modeled via a specific LP, and each hexagonal region is modeled via a different LP.

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 meeting place, 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 meeting place, it means that the estimation of their respective positions is correct. Therefore, they can form a *cluster* to explore 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 explicitly communicate to each other their current position, or they have to explicitly 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. With the ECS middleware, interactions are no longer explicit. Rather, they are implemented as in-place memory accesses, which are then fully transparently mapped to message exchanges by the middleware layer. More in detail, each LP modeling a region instantiates in its private simulation state a *presence bitmap* and an array of pointers, storing one element for each bit in the bitmap. Each bit is associated with a specific robot, and its value specifies if the robot is currently 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. Also, the pointers in the array allow to directly access, in cross-state mode, the state of LPs modeling the robots. 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 it from the region state, still thanks to the ECS support. If one robot

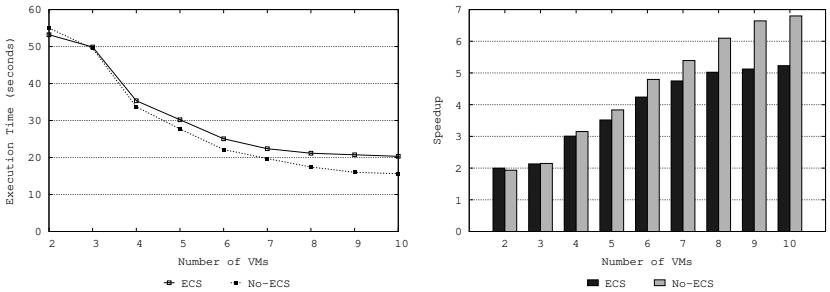

Fig. 8. Execution time and speedup for the private Cloud deploy - Multi-robot model

is found to already be in the region, then the newly entering robot simply "merges" its view of the environment. This is easily done by relying on direct access to the other robot's state—exploiting the aforementioned array of pointers. Additional details on how this model has been coded using the C programming language and the ECS support are provided in the Supplemental Material section.

In our simulations we used 484 LPs to model individual regions, and their evolution along simulation time, and 4 LPs to model the robots exploring the overall space. An important point to note for this model is that the presence bitmap and the associated array of pointers, which are kept within the state of each LP modeling a region, represent the target data structures for cross-state accesses to the region state. This information fits in a single OS page. Also, the state of an LP modeling a robot, which can be targeted in a cross-state access by an LP modeling another robot, fits within an OS page as well. Therefore, prefetching policies are essentially irrelevant for the run-time dynamics of this model—a single OS page is accessed within the state of some (remote) LP by a cross-state event, which is therefore the only page transferred across VMs if the access is a remote one. Overall, given that the minimal granularity of data transfer in our ECS middleware is a single OS page, for this model we will always have:

$$minorFaults_{NP} = minorFaults_{CP} = minorFaults_{SP}. \tag{3}$$

Hence, none of the inequalities in Equations (1) and (2) can ever be verified, leading the state machine driving the selection of the prefetch mode to naturally persist into the no-prefetch state. In other words, this model is not prone to prefetch-based optimization. This happens not because of a too articulated access pattern but rather because of the page-fitting granularity of the state information accessed by cross-state events. Hence, it looks as a test case representative of scenarios where the ECS middleware has no chance at all to proactively drive data move across VMs.

In Figure 8 (left) we report the variation of the execution time (average over 10 runs) for the multi-robot model while varying the number of VMs deployed on the private Cloud computing platform 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—based on coding the model with explicit interactions accross the LPs—and the speedup over the sequential execution, shown in Figure 8 (right), is essentially linear up to 7 VMs. On the other hand, with larger VM counts the non-ECS version scales slightly better. However, this improved scalability comes at the cost of a more complex imlementation, motivated by the need for coding the event-handler in pure data separation across the LPs, which imposes the use of specific APIs for making the LPs interact with each other explicitly.

5.2.2 Results with the Data-Store Model. As a second benchmark application, we exploited a variation of the NoSQL data-store simulation model provided in [13]. It is based on distributed/replicated cache servers, each handling a subset of the whole set of keys in the entire data-set. Any cache server is modeled by an individual LP, and the set of keys handled by the LP is implicit—it is not explicitly stored in the state of the LP—since it corresponds to a given interval of keys assigned to the LP at simulation startup. We consider a model where atomicity of the distributed transactions is ensured by running the Two-Phase Commit (2PC) protocol across the nodes keeping keys that belong to the write set of the transactions. In this model, the simulated transaction coordinator needs to schedule the arrival of a *prepare request* event to the involved cache servers, which needs to carry information about the write sets of the transactions. These sets may entail hundreds of data-item keys, and are populated by the coordinator while simulating the execution of a batch of transactions. 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 sets as the payload of the *prepare request*—for this configuration the programmer is in charge of explicitly coding the pack/unpack of the write sets—and another one based on ECS, where the write sets are directly accessed via pointers by the LP simulating a remote cache server (hence the *prepare request* event only needs to carry the pointer indicating where to find the information related to the simulated 2PC phase).

This model has features making it significantly different from the one used in Section 5.2.1. In particular, even though the data-set–namely, the keys—handled by an LP modeling a cache server is kept implicitly, the state of the LP is large, since we explicitly keep within it the write sets of the batch of transactions that are currently being executed. These sets cannot be kept implicitly since the keys touched by a transaction are not deterministic and are selected according to a random distribution. Each key in the set must be explicitly recorded into a buffer, and the overall size of the write sets spans multiple OS pages. Therefore this model is interesting to evaluate whether the prefetch policies we introduced in the ECS middleware can really pay off. Also, the CPU and memory requirements for processing events are larger compared to the model studied in Section 5.2.1. This allowed us to assess the scaling of performance with larger VM counts and with more powerful VMs (dual-vCPU ones), including the ones from the AWS computing platform.

The LP state is a unique memory-contiguous table, embedding some baseline data—not strictly related to the currently processed transactions—and the array that is used to keep the write sets of transactions. This array has a predefined size, corresponding to the maximum number of write operations that can be performed by the active transactions in a batch. However, given the non-deterministic number of keys touched by a transaction, this array could be exploited partially—it will be sequentially filled starting from its initial address with a number of entries that suffices for keeping the write sets of the transactions processed in the current batch. This also means that, when a cross-state event that accesses this array occurs, the number of OS pages that are actually accessed starting from the array initial address is non-deterministic.

We simulated a data-grid system with 64 nodes—with degree of replication 2 of each $\langle key, value \rangle$ pair across the cache servers—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 64. This configuration resembles scenarios where the 64 clients operate as front-end servers (co-located with the data-platform nodes) with respect to end-client applications. Also, the amount of keys touched in write mode by a batch of transactions is distributed between 1000 and 2000. As already hinted, this leads to non-determinism in the number of OS pages that are accessed by a cross-state event when running with the ECS middleware.

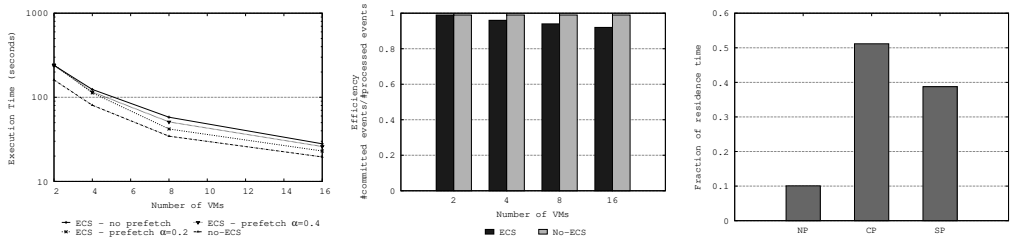

Fig. 9. Execution time, efficiency and percentage of residence times in the prefetch states with α set to 0.2 for the private Cloud deploy - Data-store model- Single-vCPU VMs

In Figure 9 we report the execution time while scaling up the number of used VMs between 2 and 16 for the case of the private Cloud computing platform. In these experiments we used single-vCPU VMs. The plots refer to two different configurations (0.2 and 0.4) of the parameter α used to drive the switch of the state machine that determines the prefetch policy—or its exclusion. Also, the time periods for determining whether to switch between states or to return back to the NP state have been set to 10^3 and 10^4 simulation time units, respectively. In the plots, we also report the execution time for the traditional PDES implementation (no-ECS) of the same simulation model, and the execution time that is achieved when using the ECS middleware with no prefetching of OS pages keeping the state of the LP hit by the cross-state event. By the data we see how the executions with the ECS middleware are able to exploit scaled up resources within the underlying platform, in fact the execution time scales down close to linearly when increasing the number of VMs. Also, the configurations where prefetch is adopted provide lower execution time with respect to the baseline configuration of the ECS middleware where prefetching is excluded. This is an indication of the capability of ECS to amortize the costs for tracking memory accesses caused by cross-state events in presence of states spanning various OS pages. Morover, the advantage of prefetching is more evident when relying on larger VMs' counts, a scenario where the simulation workloads is such that cross-state accesses more likely involve LPs hosted by different VMs. As an additional point, lower values of α lead to better tradeoffs in terms of advantages by prefetching and overhead caused by the prefetching of useless OS pages, thanks to the fact that the risk of persistent oversized prefetching is typically avoided. The execution with no-ECS provides better performance but similar scaling—when increasing the number of VMs—compared to the one with ECS, at the price of explicit pack/unpack of the write set of transactions within the application code–so as to exchange this information explicitly across LPs. Also, for this model pack/unpack operations are quite lightweight (since packing the entries of an array into the event payload does not require traversals of complex data structures to build the serialized event content). Hence the incidence of the cost for exchanging the write set explicitly when not relying on ECS is not a dominating aspect for performance. In any case, even under this somehow favorable scenario to traditional PDES, ECS-based runs still demonstrate their capability for good exploitation of the available resources.

In Figure 9 we also report a histogram that shows, for the configuration of ECS with prefetch and α set to 0.2 (which is the best performing ECS scheme), the percentage of residence-time in each state of the state machine driving prefetch operations—we recall the different states are NP, CP, and SP. By the data we see that this simulation model configuration and its memory access pattern are well captured by the CP prefetch policy, given the single table layout of the LP state and the sequential accesses to the array within the state for keeping write sets of transactions. However, SP represents a good "fallback" policy. In fact, the residence time in the SP state, although

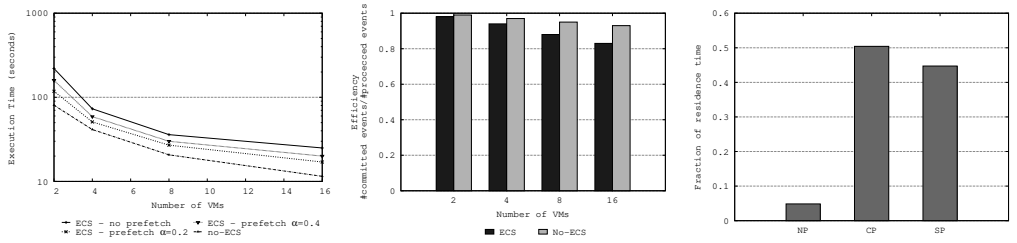

Fig. 10. Execution time, efficiency and percentage of residence times in the prefetch states with α set to 0.2 for the private Cloud deploy - Data-store model- Dual-vCPU VMs

reduced with respect to the residence time within CP, is significantly larger that the residence time in the NP state—we recall that NP is the state where prefetch is switched off. The motivation is that, even though with SP the OS pages that are transmitted in a prefetch operation are selected randomly among those keeping chunks belonging to the LP state, the likelihood to prefetch the zone of the array that does not currently keep keys to be processed in a cross-state event depends on the actual difference between the number of keys involved in the simulated 2PC phase and the array size. Therefore, large errors in prefetching useless OS pages may occur under SP only when the transactions being simulated in the current batch are characterized by randomly selected write set sizes closer to the minimum size in the admitted interval.

Still for the case of ECS with prefetch and α set to 0.2, we report in Figure 9 how the efficiency of the optimistic simulation runs varies vs the number of used VMs, compared to no-ECS. This allows to observe how the rollback pattern (and more generally the incidence of rollback) changes when relying or not on the ECS middleware. By the data we see how the execution with ECS suffers a bit more from the increase of the incidence of rollback (namely the reduction of the efficiency) when increasing the VMs' count. This is an expected behavior when considering that, under ECS, an event with smaller timestamp can lose control of the CPU—because of a cross-state access leading to the event suspension—in favor of an event with greater timestamp, hence possibly leading to more aggressive speculation along virtual time.

In Figure 10 we report the execution time while scaling up the number of used VMs between 2 and 16 for the case of dual-vCPU VMs. The observed trends are similar to the ones achieved with single-vCPU VMs, with the difference that for all the configurations, the execution time is further reduced—once fixed the number of VMs—thanks to the exploitation of the additional vCPU in each VM. These data also show how the executions based on ECS do not suffer from the increased level of actual parallelism, when comparing performance and efficiency trends to the ones observed with no-ECS—we recall that with 16 dual-vCPU machines we are actually running with a total of 32 vCPUs. As for prefetching, it appears that with increased level of intra-VM parallelism, its positive effects early appear at reduced VMs' counts. Further, also in this case lower values of α provide the highest performance advantages. Further, similarly to the case of single-vCPU VMs, the two prefetch policies included in the runtime system both appear as effective—still with a small prevalence of CP–given the definitely higher percentage of residence time in the corresponding states compared to the NP state.

The data-store model has also been run on top of the AWS computing platform, and we report in Figure 11 and Figure 12 execution times and efficiency values for both single and dual-vCPU configurations of the VMs. In the AWS platform VMs are not guaranteed to be hosted on top of a same physical machine (as instead it occurrs for our private Cloud computing platform). Hence, communication delays across VMs tend to become a more relevant factor affecting the execution

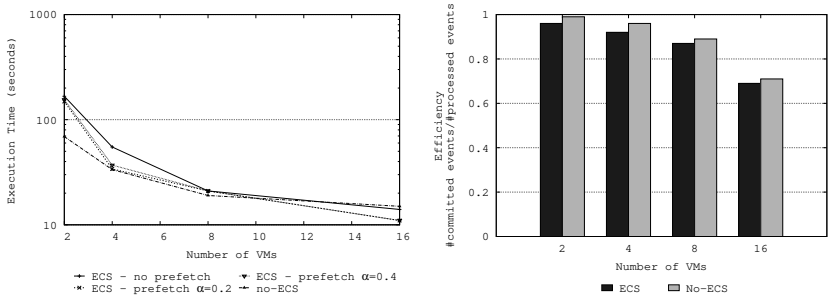

Fig. 11. Execution time and efficiency (ECS configured with $\alpha = 0.2$) for the AWS deploy - Data-store model - Single-vCPU VMs

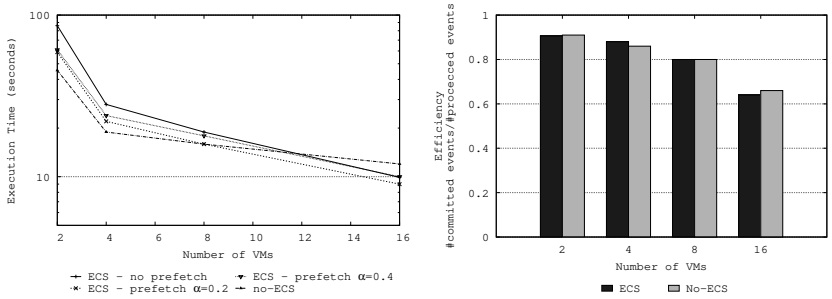

Fig. 12. Execution time and efficiency (ECS configured with $\alpha = 0.2$) for the AWS deploy - Data-store model - Dual-vCPU VMs

dynamics (e.g., the performance and the rollback pattern). Under this setting, the ECS middleware is able to deliver better performance with respect to the no-ECS case at larger VM counts essentially for two reasons. First, the efficiency values observed for the ECS and no-ECS cases are very similar (the increased comunication delay across VMs masks the slightly more agressive speculation level when running with ECS, in terms of its actual effects on the rollback generation). Second, the handling of cross-state access operations at the ECS middleware level has lower relative costs with respect to other operations, such as rollback mangement, also because of the higher impact of the incidence of rollbacks (lower efficiency) in the AWS deploy with respect to the private Cloud one. In any case, by the data, the ECS middleware is still able to well exploit increased power of the VMs (e.g. the dual-vCPU configuration of the VMs) for performance purposes compared to no-ECS.

6 CONCLUSIONS

In this article we have presented a synchronization protocol and a reference distributed middleware implementation to support the deploy of PDES simulation models implemented for shared memory on clusters of (Cloud) resources. The middleware transparently intercepts memory accesses to the state of different simulation objects by models' event handlers. If the simulation objects are running on a remote node, the middleware enforces a synchronization protocol which transparently transfers causally-consistent memory pages, therefore implementing a form of transparent speculative distributed shared memory. A decision model determines, at runtime, the best-suited amount of pages which should be prefetched upon the first remote memory access, in order to reduce the likelihood that additional synchronization is required, in order to enhance the overall performance.

Our experimental evaluation has shown that this strategy is effective under differentiated workloads and memory-access patterns. Therefore, our proposal is a viable solution to support the transparent deploy of PDES models on clusters of (Cloud) resources, without a significant burden on the model developer to explicitly code a large number of memory-based interactions across the simulation models.

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 (PADS)*. ACM, 327–336. https://doi.org/10.1145/2486092.2486134

[2] J Blomer. 2015. A Survey on Distributed File System Technology. *Journal of Physics: Conference Series* 608 (may 2015), 012039. https://doi.org/10.1088/1742-6596/608/1/012039

[3] 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)*. IEEE Computer Society, 20–28. https://doi.org/10.1145/268823.268897

[4] David Bruce. 1995. The treatment of state in optimistic systems. In *Proceedings of the 9th workshop on Parallel and Distributed Simulation (PADS)*. ACM, New York, NY, USA, 40–49. https://doi.org/10.1145/214283.214297

[5] 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. https://doi.org/10.1109/71.841745

[6] Christopher D. Carothers and Kalyan S. Perumalla. 2010. On deciding between conservative and optimistic approaches on massively parallel platforms. In *Winter Simulation Conference (WSC)*. IEEE, 678–687. https://doi.org/10.1109/WSC.2010.5679119

[7] 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. https://doi.org/10.1145/347823.347828

[8] K. M. Chandy and R. Sherman. 1989. Space-time and simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*. SCS, 53–57.

[9] 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

[10] Myongsu Choe and Carl Tropper. 2000. Flow control and dynamic load balancing in Time Warp. In *Proceedings 33rd Annual Simulation Symposium (SS)*, Vol. 18. IEEE, 9–23. https://doi.org/10.1109/SIMSYM.2000.844919

[11] Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2017. Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES. *ACM Transactions on Modeling and Computer Simulation* 27, 2 (2017), 11:1–11:26. https://doi.org/10.1145/3077583

[12] Gabriele D'Angelo and Moreno Marzolla. 2014. New trends in parallel and distributed simulation: From many-cores to Cloud Computing. *Simulation Modelling Practice and Theory* 49 (2014), 320–335. https://doi.org/10.1016/j.simpat.2014.06.007

[13] Pierangelo di Sanzo, Francesco Quaglia, Bruno Ciciani, Alessandro Pellegrini, Diego Didona, Paolo Romano, Roberto Palmieri, and Sebastiano Peluso. 2015. A flexible framework for accurate simulation of cloud in-memory data stores. *Simulation Modelling Practice and Theory* 58 (2015), 219–238. https://doi.org/10.1016/j.simpat.2015.05.011

[14] Javier Díaz, Camelia Muñoz-Caro, and Alfonso Niño. 2012. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *IEEE Trans. Parallel Distrib. Syst.* 23, 8 (2012), 1369–1386. https://doi.org/10.1109/TPDS.2011.308

[15] A. Fabbri and L. 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

[16] 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

[17] Richard M Fujimoto. 1990. Performance of Time Warp Under Synthetic Workloads. In *Proceedings of the Multiconf. on Distributed Simulation*. Society for Computer Simulation, 23–28.

[18] 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.

[19] D 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

[20] B. Kaan Gorur, Kayhan Imre, Halit Oguztuzun, and Levent Yilmaz. 2016. Repast HPC with Optimistic Time Management. In *Proceedings of the 24th High Performance Computing Symposium (HPC '16)*. Society for Computer Simulation International, San Diego, CA, USA, Article 4, 9 pages. https://doi.org/10.22360/SpringSim.2016.HPC.046

[21] Danny Hendler, Alex Naiman, Sebastiano Peluso, Francesco Quaglia, Paolo Romano, and Adi Suissa. 2013. Exploiting Locality in Lease-Based Replicated Transactional Memory via Task Migration. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*. 121–133. https://doi.org/10.1007/978-3-642-41527-2_9

[22] 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.

[23] Golden G. Richard III and Mukesh Singhal. 1993. Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. In *12th Symposium on Reliable Distributed Systems, SRDS 1993, Princeton, New Jersey, USA, October 6-8, 1993, Proceedings*. 58–67. https://doi.org/10.1109/RELDIS.1993.393473

[24] David R. Jefferson. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems* 7, 3 (jul 1985), 404–425. http://portal.acm.org/citation.cfm?doid=3916.3988

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

[26] Michael Lees, Brian S. Logan, and Georgios Theodoropoulos. 2008. Using Access Patterns to Analyze the Performance of Optimistic Synchronization Algorithms in Simulations of MAS. *Simulation* 84, 10-11 (2008), 481–492. https://doi.org/10.1177/0037549708096691

[27] Nazzareno Marziale, Francesco Nobilia, Alessandro Pellegrini, and Francesco Quaglia. 2016. Granular Time Warp objects. In *Proceedings of the 2016 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*. ACM Press, New York, New York, USA, 57–68. https://doi.org/10.1145/2901378.2901390

[28] 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

[29] Jens Müller, Sergei Gorlatch, Tobias Schröter, and Stefan Fischer. 2007. Scaling Multiplayer Online Games Using Proxy-server Replication: A Case Study of Quake 2. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing (HPDC '07)*. ACM, New York, NY, USA, 219–220. https://doi.org/10.1145/1272366.1272399

[30] 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* 44, 6 (2016), 1200–1247. https://doi.org/10.1007/s10766-016-0429-2

[31] Alessandro Pellegrini and Francesco Quaglia. 2014. Wait-free Global Virtual Time computation in shared memory Time-Warp systems. In *Proceedings of the 26th International Conference on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE Computer Society, Paris, France, 9–16. https://doi.org/10.1109/SBAC-PAD.2014.38

[32] Alessandro Pellegrini and Francesco Quaglia. 2017. A Fine-Grain Time-Sharing Time Warp System. *ACM Trans. Model. Comput. Simul.* 27, 2 (2017), 10:1–10:25. https://doi.org/10.1145/3013528

[33] Alessandro Pellegrini and Francesco Quaglia. 2019. Cross-state events: A new approach to parallel discrete event simulation and its speculative runtime support. *J. Parallel Distrib. Comput.* 132 (2019), 48–68. https://doi.org/10.1016/j.jpdc.2019.05.003

[34] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2011. The ROme OpTimistic Simulator: Core Internals and Programming Model. In *Proceedings of the 4th ICST Conference of Simulation Tools and Techniques (SIMUTools) (SIMUTools)*. ICST, 96–98. https://doi.org/10.4108/icst.simutools.2011.245551

[35] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2015. Autonomic State Management for Optimistic Simulation Platforms. *IEEE Transactions on Parallel and Distributed Systems* 26, 6 (June 2015), 1560–1569. https://doi.org/10.1109/TPDS.2014.2323967

[36] 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

[37] Paolo Romano, Luís E. T. Rodrigues, Nuno Carvalho, and João P. Cachopo. 2010. Cloud-TM: harnessing the cloud with distributed transactional memories. *Operating Systems Review* 44, 2 (2010), 1–6. https://doi.org/10.1145/1773912.1773914

[38] Tim Stitt. 2010. An Introduction to the Partitioned Global Address Space (PGAS) Programming Model. (2010).

[39] Vinoth Suryanarayanan and Georgios Theodoropoulos. 2013. Synchronised Range Queries in Distributed Simulations of Multiagent Systems. *ACM Trans. Model. Comput. Simul.* 23, 4, Article 25 (Nov. 2013), 25 pages.

[40] Vinoth Suryanarayanan, Georgios Theodoropoulos, and Michael Lees. 2013. PDES-MAS: Distributed Simulation of Multi-Agent Systems. In *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013*. 671–681. https://doi.org/10.1016/j.procs.2013.05.231

[41] Srikanth B. Yoginath and Kalyan S. Perumalla. 2013. Optimized hypervisor scheduler for parallel discrete event simulations on virtual machine platforms. In *Proceedings of the 6th International ICST Conference on Simulation Tools*

and Techniques (SimuTools). 1–9.
[42] 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.

Received December 2018; revised XXX 2019; accepted October 2019