ISSN 2281-4299

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Transparent Distributed Cross-State Synchronization in Optimistic Parallel Discrete Event Simulation

Matteo Principe
Alessandro Pellegrini
Francesco Quaglia
Bruno Cruciani

Technical Report n. 12, 2017

Transparent Distributed Cross-State Synchronization in Optimistic Parallel Discrete Event Simulation

Matteo Principe[*]
matteo.principe92@gmail.com

Alessandro Pellegrini[*]
pellegrini@dis.uniroma1.it

Francesco Quaglia[†]
francesco.quaglia@uniroma2.it

Bruno Ciciani[*]
ciciani@dis.uniroma1.it

December 2017

Abstract

In this report we tackle transparent deploy and seamless execution of sequentially-coded Parallel Discrete Event Simulation (PDES) models on distributed computing architectures. We present an innovative distributed synchronization protocol which allows, in conjunction with ad-hoc Operating System memory management facilities, to access the simulation state of any concurrent Logical Process (LP) running on any node of the distributed computing environment, as if it were locally hosted by a unique node—more specifically, by a unique address space. By relying on our facilities, the simulation model developer is not required to implement neither explicit message passing, nor to rely on annotations or specific programming constructs. He can simply code the accesses to the LPs' states in place (e.g. via pointers), which significantly simplifies the software development process. The burden of synchronization and correct handling of these accesses is demanded from our user-space and kernel-space runtime environment. Our proposal targets Linux on x86_64 systems and has been integrated within the ROOT-Sim open-source optimistic simulation platform, although its design principles, and most parts of the developed software, are of general relevance.

Keywords: *Distributed Simulation, High-Performance Computing, PDES, Programming Models*

[*]DIAG, Sapienza, University of Rome
[†]DICII, University of Rome "Tor Vergata"

1 Introduction

The advent and large diffusion of technologies such as multi-core platforms and the Cloud is strongly impacting the way advanced simulation environments are built. Altogether, such an impact is even antithetical, because multi-core systems tend to exalt computing modes based on data sharing, while the Cloud offers opportunities for low-cost resources (e.g., spot instances) that can be exploited as a cluster according to distributed-memory oriented programming models.

In this report our focus is on bridging the two worlds—computing on shared vs distributed memory platforms—for the case of Parallel Discrete Event Simulation (PDES). The goal is to support transparent deploy on distributed clusters of PDES models implemented in a fully sequential style, which can therefore leverage shared-memory accesses of the simulation state. In particular, we present the design and implementation of software facilities (based on a combination of user space and kernel space code) that allow PDES models coded by explicitly relying on the data-sharing paradigm proper of sequential implementations to be transparently deployed and executed seamlessly on distributed memory clusters of (multi-core) machines. Our proposal is important also in the light of the fact that distributed computing platforms made up by multi-core machines are fundamental to target the transition from petascale to exascale simulations. Indeed, the recent architectural trend is to rely on larger clusters of parallel computing nodes, as the way to overcome both the *power wall* [1] and the *memory wall* [2], which have posed strict limitations on what can be done with current off-the-shelf computing systems.

To make our objective and the contribution by this report clearer, let us recap the foundations of the PDES paradigm and some recent innovations brought into PDES due to its reshuffle towards shared-memory multi-core computing. In PDES, a complex simulation model is partitioned into distinct portions, known as simulation objects or Logical Processes (LPs), which mimic different parts of the overall simulated system [3]. LPs are concurrently scheduled for execution, so as to enable speedup while processing the simulation model. Further, the traditional specification of PDES has been based on disjointness of the accesses to the state of the LPs while processing simulation events. Interactions among the different portions of the model were indeed based on explicit event exchange, mostly supported via message passing. This perfectly fitted the scenario where model execution parallelism was based on deploying the application on top of distributed memory clusters. On the downside, this way of coding PDES models has historically forced the programmer to separate at implementation time the accesses to slices of the simulation model state, each one representing an individual LP.

More recent research efforts have been devoted to rethinking the PDES paradigm and its support, for better exploitation of shared-memory accesses on multi-core machines, particularly by the side of making the models' programming job more flexible. Indeed, for several application classes (e.g., demographic agent-based models [4]), enabling the application modules to directly access the state of some concurrent LP, while processing a simulation event at another LP, can extremely simplify the task of coding a model. Such a capability avoids the need for explicitly coding cross-LP event scheduling each time an action at some LP depends on the current state of some other LP. Rather than discovering the information via query events, it can be directly read from the state of the target LP by simply exploiting pointers to the state. Furthermore, beyond querying, direct updates on the target LP state can take place. Overall, while PDES has been historically characterized by event-based synchronization across the LPs, new shared-memory oriented incarnations of the PDES paradigm have introduced the concept of Event & Cross-State (ECS) synchronization [5], based on the mixture of event exchanges and direct accessibility of the whole set of the states of the concurrent LPs by a module of the application.

With the software architecture that we present in this report we transparently materialize ECS on top of distributed memory systems. Thus we enable the exploitation of any kind of distributed (Cloud) resource for running the simulation model, while not sacrificing the flexibility of shared-memory accesses offered to the programmers by the ECS paradigm. Our proposal has also the advantage of fully transparently enabling speculative (optimistic) processing—and rollback-based causal consistency—of the concurrent LPs [6], an approach that is recognized as a core building block for actual scalability of the model execution.

In our design we target the Linux operating system, offering new memory management capabilities able to:

- detect the materialization of cross state accesses among LPs hosted on different machines, based on pointer de-referencing in shared-memory (sequential-style) application coding;
- implement an approach where the accessing LP transparently gains a lease on the logical pages associated with the state of the target LP, at the snapshot corresponding to the the correct simulation time—the one of the event that triggered the access;
- detect on the fly the class of machine instructions actually accessing the target LP state, so as to pre-fetch and locally materialize the suited number of pages for finalizing the access to the remotely hosted LP, and writing back onto the hosting node only dirty pages upon the end of the cross-state interaction.

Our software architecture has been integrated within the ROOT-Sim open-source PDES runtime environment [7] and is freely available for download[1]. Experimental data reported in this report show the viability of our proposal.

The remainder of this report is structured as follows. In Section 2 we discuss related work. Section 3 presents our reference system model. In section 4 we discuss the innovative distributed architecture to support ECS, which is experimentally evaluated in Section 5.

2 Related Work

The issue of bypassing state disjointness in PDES concurrent objects has been dealt with by several studies. The work in [8] discusses how state sharing might be emulated by using a separate LP 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 in order to cope with read/write operations occurring at different logical times, and to avoid unneeded rollbacks of the centralized server in case of optimistic synchronization. This is an approach similar to the one in [9], where a theoretical presentation of algorithms to implement a Distributed Shared Memory mechanism is provided in terms of protocols to keep replicated instances of a variable coherent. The above approaches are different from what we propose given that they essentially rely on a message-passing API to nest the read/write access to shared variables into the application code. Instead, we support accesses as in-place specified (e.g. via pointer de-referencing) within the shared memory oriented code implementation, fully demanding from the innovative operating system memory management facilities their handling. This has also the advantage of optimizing the execution path of memory management operations depending on whether the cross state access hits a locally-hosted vs a remote LP. Also, in our proposal sharing is not limited to a particular memory slice (such as the state image of the centralized server), while we allow access, and hence sharing, of any memory buffer representing a portion of the whole simulation model state—this is intrinsically linked to the possibility to navigate memory across concurrent objects via pointers which we provide.

In [10] the notion of *state query* is introduced. A LP needing a portion of the state which belongs to a different object can issue a query message to it, and wait for a reply containing the suitable value. In case this value is later detected to be no longer valid, an anti-message is sent to invalidate the query. This approach still relies on message passing, and is not transparent to the application programmer.

Other works [11–13] have proposed approaches for shared accesses across simulation objects, which are in some cases based on in-place operations, transparently handled by the underlying run-time environment. However,

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

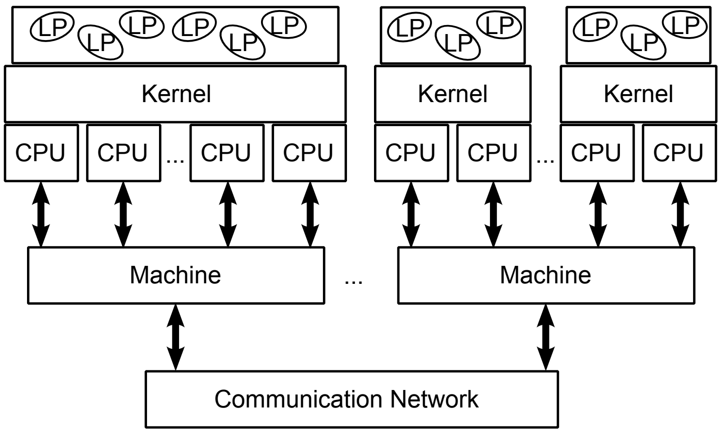

Figure 1: Reference System Model's Organization

these proposals are limited to managing accesses that are confined to the memory handed by a single multi-core machine. Instead, our goal it to seamlessly enable such kind of accesses on distributed memory systems.

In the context of the High-Level-Architecture (HLA), proposals for supporting shared states can be found in [14, 15]. They are based on a middleware component which relies on a timestamp-ordering messaging approach for implementing a request/reply protocol. These approaches are targeted at the conservative synchronization protocol, where there is no need to detect and handle causality violations, while we target optimistic synchronization and in-place accesses to the object states.

3 Reference System Model

A high-level organization of our reference system model is depicted in Figure 1. A parallel and distributed discrete-event simulation is supported by a set of (possibly non-homogeneous) processing units, scattered across any number of machines (i.e., computing nodes). On each computing node, any number of *simulation kernel instances* can be running. These instances are developed according to the symmetric multi-threaded paradigm, introduced for the first time in [16], where shared memory is used to support intra-kernel synchronization. Distributed communication is supported by some network interconnection.

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

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

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

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

4 Distributed Event Cross-state Synchronization

4.1 Basics

Similarly to the original proposal in [5], our distributed ECS architecture is based on two orthogonal facilities which are transparently offered by the simulation platform. On the one hand, while simulation events are being executed, the platform is able to *detect* that a LP is accessing the state of another LP, possibly hosted by a remote simulation kernel instance. At the same time, the platform is able to enforce a (distributed) protocol to *synchronize* the Local Clocks of the LPs involved in an ECS synchronization, so as to allow them to observe a consistent view on the simulation state.

In our organization, *cross-state access detection* is provided by innovative kernel-level facilities, which let different worker threads of the platform to share the same physical pages although with different access privileges. Therefore, a page fault upon accessing the simulation state of a different LP is the initiation of an ECS synchronization, as it will be later discussed. At the same time, *LP synchronization* is enforced by relying on a (distributed) communication protocol, based on the notion of *control messages*. A control message is a message exchanged across two different LPs, in a way

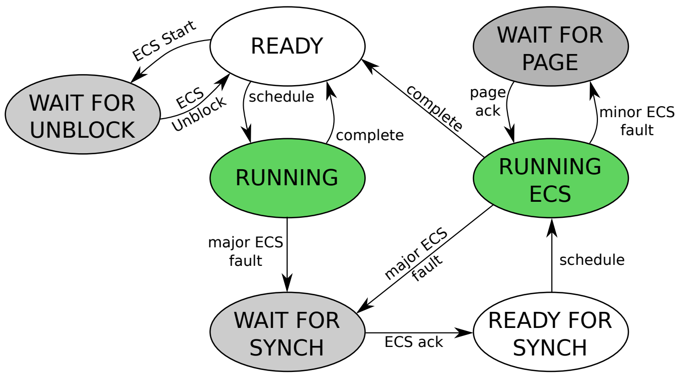

Figure 2: LP State Machine.

completely similar to event transfer. Nevertheless, with one single exception, a control message is not incorporated into the receiver's event queue, as they are associated with ephemeral state transitions which must not be replayed upon a rollback operation.

Correctness of the whole simulation is guaranteed by two facts: i) the execution of an event by a LP can be *suspended*; ii) every LP is always in an execution state according to the state machine depicted in Figure 2, which allows the simulation platform to correctly interpret the system events and control messages which target every LP.

As for point i) above, we rely on User-Level Threads (ULT), namely CPU contexts which can be saved and restored at any time instant by a worker thread $t \in TSet_k$. In particular, to give control to a LP, the worker thread in charge of it changes its CPU context, allowing the execution of the event to take place in an isolated environment, which has also its own stack. In this way, whenever the simulation platform takes back control, it might determine that the event's execution has to be temporarily suspended, and it deschedules the running LP (i.e., it restores the CPU context related to the worker thread running in *platform mode*). Later, the worker thread can decide to resume the execution of the suspended event, and this is done by simply restoring the LP's CPU state. Having a separate stack for every LP within a single worker thread (which has its own system stack) ensures the correctness of the preëmptive event execution. For a thorough technical description of the approach used to realize this facility, we refer the reader to [17].

With respect to point ii) above, the state machine reported in Figure 2 has three different types of states: *blocked states* (gray-shaded) are associated with a LP which has been descheduled while executing an event, thanks to the ULT facility; *ready states* (white-colored) are associated with LPs which can be activated, either to start processing a new event, or to resume the execution of a preëmpted event; *running states*, which are associated with LPs currently executing an event. This organization allows to implement the *smallest-timestamp first* scheduling strategy [3] of each worker thread quite easily, given that only LPs in a ready state can be activated. The transitions across the different states are related to two main kind of events: some are associated with the aforementioned *cross-state access detection*, others with the actual *LP synchronization*. We will thoroughly describe these transitions later.

4.2 Memory Management

In order to support cross-state access detection, the runtime environment must enforce a memory management policy which allows in a simple way to map a LP to a given memory address, and viceversa. This is particularly important given that we must discriminate between memory accesses which target the simulation states of both local and remote LPs.

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

We have therefore resorted, in a way similar to what has been proposed in [18], to a *deterministic memory map manager*. In particular, in accordance to the original proposal in [5], each LP is associated with a 1 GB *memory stock*. As illustrated in Figure 3, the base address of this stock is deterministically computed by every simulation kernel instance. In this way, all simulation kernel instances map LPs stocks to a same contiguous region of the virtual address space, where the stocks are uniquely associated with an address range which does not overlap.

Given that a given simulation-kernel instance manages a pre-defined set of LPs, thanks to its worker threads, at simulation startup these memory stocks are delivered to a fine-grained memory manager, such as the one presented in [19], which ensures that the simulation model's memory requests can be served thanks to traditional APIs, such as `malloc` or `new`.

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

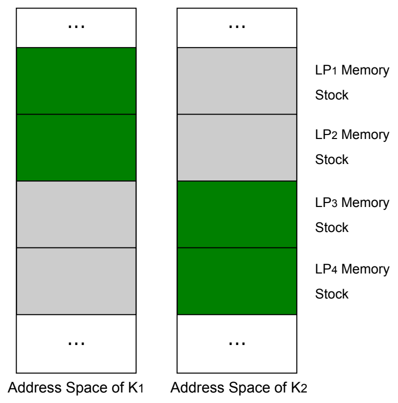

Figure 3: LP Memory Map Organization

4.3 Kernel-Level Support

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

4.3.1 Explicit Interaction

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

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

The memory map depicted in Figure 3 is allocated so that the page table respects an important invariant. We allocate LPs' memory stocks so that the whole GB of memory is aligned to one single entry in the Page Directory Pointer (PDP) table. In this way, any access to any physical page

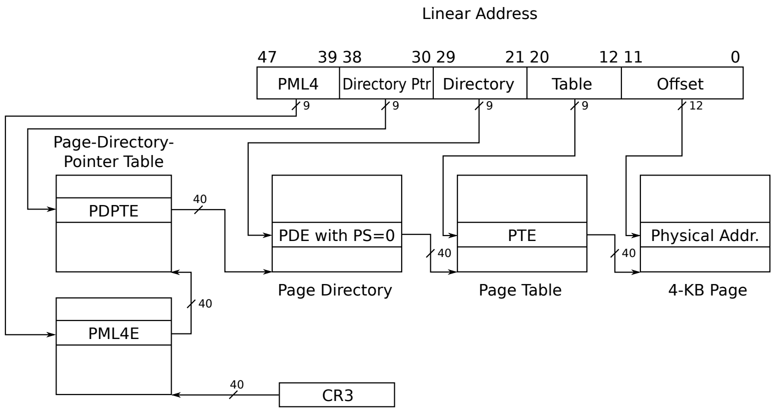

Figure 4: The paging scheme in x86_64 processors.

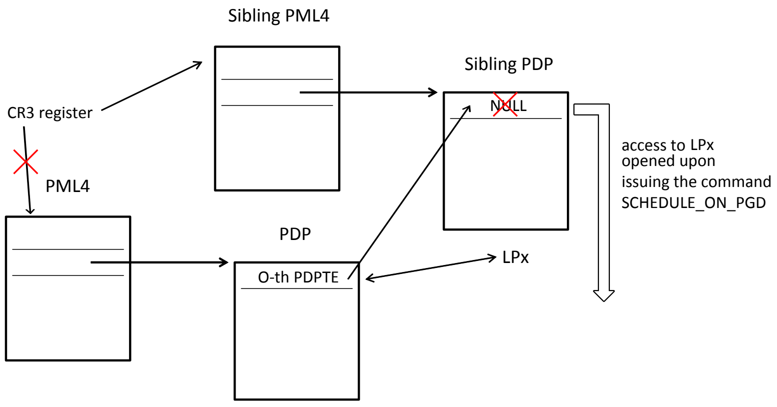
Sibling PML4
Sibling PDP
NULL
CR3 register
access to LPx opened upon issuing the command SCHEDULE_ON_PGD
PML4
PDP
O-th PDPTE
LPx

Figure 5: ECS Schedule Example

related to the simulation state of a LP can be immediately mapped to the actual LP thanks to the PDP entry used in the virtual-to-physical address resolution. Therefore, thanks to the deterministic memory allocation scheme enforced, the payload of the `SET_VM_RANGE` `ioctl` command is simply the initial address of the first memory stock reserved for LP_0, and the total number of LPs.

To actually determine when a LP is accessing the state of a different LP, worker threads inform the kernel module what is the LP which will be activated for event execution via another `ioctl` command named `SCHEDULE_ON_PGD`. This command activates a kernel-level logic implemented in the module which installs a *sibling page table* on the `CR3` register of the CPU core running the worker thread. In particular, the invocation of the `SCHEDULE_ON_PGD` command puts in place the policy illustrated in Figure 5. The sibling page table is constructed by cloning the PML4 table associated with the virtual memory of the whole process—this can be easily retrieved by the module from `current->mm->pgd`—and by cloning the PDP tables which point to the simulation state of any LP, be it both local and remote. These cloned PDP tables are zeroed, except for the entry associated with the currently-scheduled LP (notified via the `ioctl` call) so that whenever an access is made towards a different LP's simulation state, it generates a memory fault.

Having different sibling PML4 tables associated with the different concurrent worker threads leads to the possibility to concurrently dispatch and execute different LPs—this is done by having each worker thread opening the access to the stocks associated with the object it is currently dispatching—while still having the possibility to determine whether any of the dispatched objects is confining its memory references within its own stocks. The assumption underlying this type of organization is that, when there is the need for opening access to a given stock, the corresponding memory management information is already present in the corresponding PDP entry of the original page tables. This is not guaranteed by simply validating virtual memory addresses via `mmap`, which leaves memory into the empty-zero state. To overcome this problem, when we initialize the memory map depicted in Figure 3, beyond calling `mmap`, we also explicitly write a null byte into one single virtual page of the stock. In this way, the Linux kernel traps the access to empty-zero memory and allocates the whole chain of page tables for managing the pages within the stock (although a single one of these pages is really allocated). This guarantees the existence of the PDP entry associated with the stock, to be filled into the corresponding sibling PDP entry upon dispatching the LP owning the stock. We note that relying on more traditional facilities, such as `mprotect` would not be viable. Indeed, this would setup policies which are enforced for the whole process, while our approach allows different threads within the same process to observe different memory access privileges, at a very negligible cost.

4.3.2 Implicit Interaction

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

Once the ECS fault handler is activated, it first checks whether the handler is activated to resolve a minor page fault from kernel space (**F1**) or

Algorithm 1 ECS Page Fault Kernel Handler

```
procedure FaultHandler(pt_regs* $regs$)
    if $current \to mm$ = NULL then                               ▷ F1
        DoPageFault( )
        return
    if $current \to pid$ is not registered then                   ▷ F2
        DoPageFault( )
        return
    $target \leftarrow$ ReadCR2( )
    if PML4($target$) not in LP range then                        ▷ F3
        DoPageFault( )
        return
    else
        if PDP($target$) = NULL then                              ▷ F4
            $fault\_type \leftarrow Major$
        else
            if GetPteStickyBit($target$) then                     ▷ F5
                $fault\_type \leftarrow Minor$
                SetPresenceBit($target$)
            else
                if ¬GetPresenceBit($target$) then                 ▷ F6
                    DoPageFault( )
                    if GetPdeStickyBit($target$) then
                        $fault\_type \leftarrow Minor$            ▷ F7
                        SetPageStickyFlag($target$)
                    else
                        return
                else                                              ▷ F8
                    $fault\_type \leftarrow AccessChange$
                    SetPagePrivilege($target$, WRITE)
    Switch to the original Page Table                             ▷ F9
    Copy to userspace fault information
    Push on userspace stack $regs \to ip$
    $regs \to ip \leftarrow$ EcsHandler                           ▷ F10
```

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

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

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

Conversely, we exploit the organization of a Page-Table Entry (PTE), which is depicted in Figure 6, in the original memory view. In particular, we scan all PTE entries which can be reached starting from the PDE entry associated with the given remote LP towards which the scheduled one is synchronizing. All non-null PTE entries, which are thus associated with an actual materialized page, have the *presence bit* (bit 0) set to 1, to indicate that the Page Base Address is a valid (physical) base pointer for the page. We explicitly force the presence bit to zero, thus generating an additional artificial memory fault whenever such a page (installed by the userspace

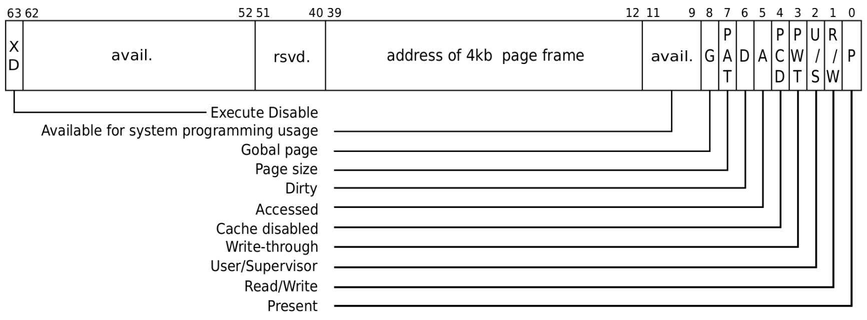

Figure 6: Page-Table Entry (4KB Page)

handler) is accessed. To discriminate whether a fault is artificial or not, due to the above-described scheme, before clearing the present bit we set bit 9 in the same PTE entry. This is an *available bit* (which we use as a sticky bit) i.e. a bit which can be used by the LKM to implement additional facilities not supported by the firmware. While performing this action, we similarly set one available bit in the PDE entry, to mark the whole memory stock as associated with a remote LP.

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

The check at **F6** is important, as it covers as well an additional case. When a LP accesses a remote page in read mode, we explicitly prevent the possibility to access the local copy of the page installed by the userspace handler in write mode by setting bit 1 of the associated PTE to zero. This bit (see Figure 6) is the *read/write bit* which, when set to zero, generates a memory fault when the page is accessed in write mode. In this case (**F8**) we

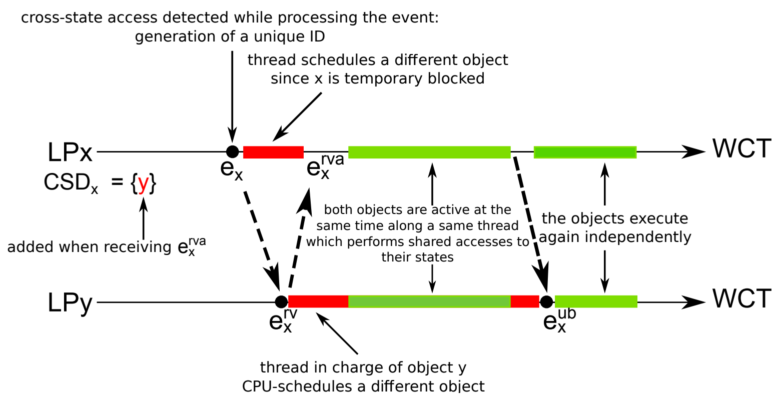
cross-state access detected while processing the event: generation of a unique ID
thread schedules a different object since x is temporary blocked
LPx
CSD$_x$ = {y}
added when receiving e_x^{rva}
e_x
e_x^{rva}
both objects are active at the same time along a same thread which performs shared accesses to their states
the objects execute again independently
WCT
LPy
e_x^{rv}
e_x^{ub}
WCT
thread in charge of object y CPU-schedules a different object

Figure 7: LP Synchronization Scheme

explicitly set back this bit to 1, enabling the possibility to write the page, and deliver an ECS Access Change Fault to the userspace handler. This is an important aspect, as we will show how later how this can optimize the finalization of the ECS protocol.

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

4.4 LP Synchronization Protocol

Before entering in the details of the userspace ECS handler, we discuss the (distributed) protocol to synchronize two LPs whenever a cross-state access is detected. Synchronization is supported by control-message passing among the involved nodes. The basic scheme is depicted in Figure 7.

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

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

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

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

4.5 Userspace ECS Management

When the LKM notifies the runtime environment that two LPs have to be ECS-synchronized, the handler depicted in Algorithm 2 is activated. This

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

[3]For a thorough description of the rollback strategy and all its implications on liveliness and correctness of the approach, we refer the reader to [19].

Algorithm 2 Userspace ECS Handler

```
1:  procedure EcsHandler($type$, $info$)
2:      if $type = Major$ then                                        ▷ H1
3:          $ECS\_mark \leftarrow$ generate_mark( )
4:          Send(RENDEZVOUS, $info.targetLP$, $currentLVT$)
5:          $LP\_state \leftarrow$ WAIT_FOR_SYNCH
6:          $CSD \leftarrow CSD \cup \{info.targetLP\}$
7:          Deschedule( )
8:      else if $type = Minor$ then                                   ▷ H2
9:          $disasm \leftarrow$ Disassemble($info.rip$)
10:         $write\_mode \leftarrow disasm.write$
11:         $page\_addr \leftarrow$ BaseAddr($info.target$)
12:         $pages \leftarrow$ PgCount($info.target$, $disasm.span$)
13:         if $write\_mode$ then
14:             AddToWriteList($page\_addr$, $pages$)
15:         else
16:             AddToReadList($page\_addr$, $pages$)
17:         Send(PAGE_LEASE, $info.targetLP$, $currentLVT$)
18:         $LP\_state \leftarrow$ WAIT_FOR_PAGE
19:         Deschedule( )
20:     else if $type = AccessChange$ then                            ▷ H3
21:         $page\_addr \leftarrow$ BaseAddr($info.target$)
22:         AddToWriteList($page\_addr$, 1)
```

handler performs different actions depending on the type of ECS fault which is notified by the LKM.

The ECS Major Fault case (**H1**) is associated with the initiation of the (distributed) protocol described in Section 4.4. First, a system-wide unique mark is generated, and a rendezvous start message is sent to the LP keeping the portion of the simulation state which is being accessed by the currently-scheduled event. The id of the target LP is delivered by the LKM, as it is uniquely associated with the PDP entry related to the faulting memory address. The running LP then enters the Wait For Synch state, and the target LP is added to the CSD of the running LP. Finally, the running LP is descheduled thanks to the ULT facilities described before. In this way, the running LP will never be activated until the rendezvous ack is received, as previously described.

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

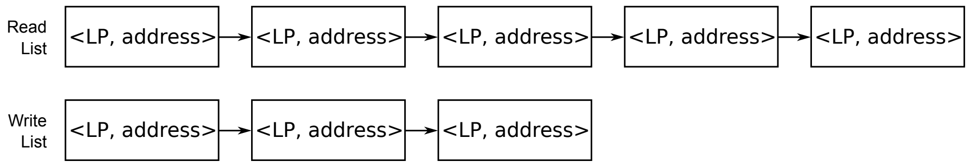

Figure 8: Page Touch Lists

userspace handler by the LKM, together with a snapshot of the relevant CPU registers as observed by the faulting instruction.

The disassembler[4] provides several relevant pieces of information regarding the faulting instruction. Among these, we can determine whether the instruction is accessing in read or write mode, and the size of the memory access. The latter information is used in conjunction with the target memory address where the instruction has faulted notified by the LKM to determine the base address of the first (remote) page which has to be transferred to the local node, and the number of pages. This information is sent to the destination LP as an additional control message, named `PAGE_LEASE`, before putting the LP in the blocked Wait For Page state. Once this control message is received at the target LP. Since the target LP is already in a blocked state, we are actually acquiring a *lease* on the pages, having the LP which originated the ECS synchronization keep a temporary master copy of the content of that portion of the state. These pages can be safely installed into the local address space, thanks to the non-overlapping organization of the memory map manager depicted in Figure 3.

To keep track of what pages have been leased by an LP, we maintain two *page-touch lists*. One list is associated with pages accessed in read mode, which we refer to as *read list*, while the other (the *write list*) is associated with pages accessed in write mode. The two lists keep as well the id of the LP whose the original page belongs to, as depicted in Figure 8. This is due to the fact that during the execution of an event, a LP is allowed to synchronize with any number of LPs, thus we must keep track of the ownership of each page. To reduce the complexity of this management, each node in the list is associated with a PTE entry, where a bitmap of 512 bits (one for each page) is used to determine whether the corresponding page has been locally acquired or not.

When accessing a page in read mode, the LP has already acquired a lease on it and a copy of the page content is already installed in the local address space. Since the underlying operating system has granted access in read mode only, once the event handler accesses the same page in write mode, a new fault is detected. Nevertheless, this latter fault can be resolved locally. When activating the userspace handler for an ECS Access Change

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

Fault (**H3**), the LKM has already upgraded the access privilege to write mode. The userspace has only to move the page from the read list to the write list, in the corresponding PTE node.

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

4.6 Memory Reclaim

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

This operation is supported by having one single worker thread at each node invoke a sequence of `munmap`/`mmap` for every memory stock associated to remote LPs. In this way, we instruct the underlying operating system to release all memory pages which have been materialized during the execution of remote cross-state synchronizations. It is fundamental to execute this operation in isolation, i.e. when no other thread has any operation related to a remote ECS still pending. In our implementation, we have resorted to a periodic check, with a period of around 30 seconds, where all threads notify in shared variables whether they have pending remote synchronization, and an additional shared variable is used to delay the initiation of a remote ECS if a memory reclaim phase is in progress.

5 Experimental Assessment

We have integrated the proposed architecture within ROOT-Sim, an open source C/MPI-based simulation platform targeted at POSIX systems [7], which implements a general-purpose parallel/distributed simulation environment relying on the optimistic synchronization paradigm. ROOT-Sim offers a very simple programming model based on the classical notion of simulation-event handlers, to be implemented according to the ANSI-C standard, and transparently supports all the services required to parallelize the execution.

As the benchmark application, we have used a multi-robot exploration and mapping simulation model, according to the results in [20]. In this model, a group of robots is set out into an unknown space, with the goal of fully exploring it, while acquiring data from sensors (e.g., cameras, lasers, ...) which are used to map the environment. The robots are equipped with enough processing power to elaborate the sensors data online (thus, the map is constructed during the exploration), so as to allow them to rely on the acquired knowledge to drive the exploration in a more efficient way. Specifically, whenever a robot has to make a decision about which direction should be taken to carry on the exploration, it is done by relying on the notion of *exploration frontier*. By keeping a representation of the explored world, the robot is able to detect which is the closest unexplored area which it can reach, computes the fastest way to reach it and continues the exploration.

The robots explore independently of each other until one coincidentally detects another robot. Whenever two robots enter a proximity region, they perform three different actions: i) they use their sensors to estimate their mutual physical position—recall that they are just in *proximity*; ii) they verify the goodness of their position hypothesis by creating a rendez-vous point (not to be confused with rendez-vous control messages related to the ECS protocol) in the explored part of the region, and trying to meet again there; iii) if the hypothesis is verified, they exchange the data acquired during the exploration, thus reducing the exploration time and allowing for a more accurate decision of the actions to be taken. Additionally, in case step ii) succeeds (i.e., the robots actually meet in the rendez-vous point), it means that the estimation of their respective position is correct. Therefore, they can form a *cluster*, i.e. they can start exploring the environment in a collaborative way. This collaborative exploration takes place in two different ways. On the one hand, they jointly define (by relying on *cost* and *utility* functions, as defined in [20]) their next exploration targets, so that they can minimize the time required for a complete environment exploration. On the other hand, they might decide to make a *guess* about the position of other robots (the total number of which is known) which are not part of the cluster yet. In the latter case, one of the robots (the one for which the utility/cost ratio is convenient) targets the hypothesized position. If a robot is found there, the aforementioned steps are carried out, so as to increase the knowledge of the environment.

This model is a good test-case for exploiting the programming paradigm and facilites offered by ECS. In our implementation, we rely on two different types of LPs: *robots* and *regions* of the exploration environment, which is represented as a square region divided into hexagonal cells. This choice allows us to define a meaningful mobility model for the robots, and at the same time allows us to define proximity regions which are used by the robots to detect the presence of other ones in the nearby.

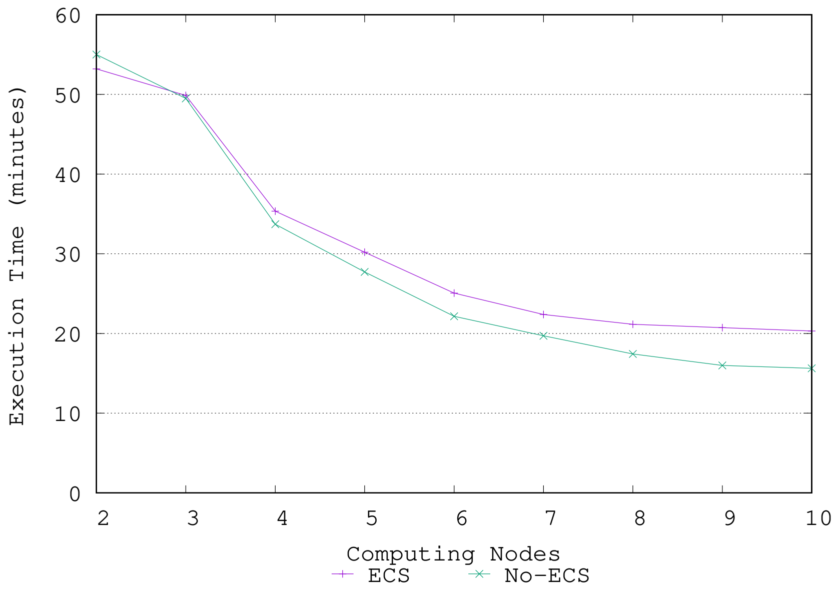

Figure 9: Simulation Throughput

We have run experiments on a cluster of virtualized nodes, composed of up to 10 Virtual Machines (VMs). The VMs are hosted by the VMware Workstation hypervisor (version 10.0.4 build-2249910) hosted on top of a HP ProLiant server equipped with 100 GB of RAM and 8 AMD Opteron 6376 CPUs running at 2.6 GHz. Each one has four cores (for a total of 32 physical cores). We have installed Debian 6.0.7 with Linux kernel 2.6.32-5. Each VM has two virtual cores and 8 GB of memory, thus overall mimicking a cluster of mid-range computers. All available virtual cores are used by ROOT-Sim to carry out the simulation.

We have configured our model to use 4 robots to explore a region composed of 484 hexagonal cells. We have used two different implementations. The first one (which we refer to as "ECS" in the results) is implemented by relying on memory pointers used both to exchange data across robots, and to register one robot within a cell—upon initialization, all cells register their states into a shared array which is replicated across all computing nodes. This is the implementation that triggers the transparent distributed synchronization protocol presented in this report. The second one (which we refer to as "No-ECS" in the results) relies on traditional message passing to implement the same logic. in this latter version, when a robot exchanges the information on the explored portion of the map, the data are explicitly marshalled into the event's payload. As mentioned, this is an added complexity on the model's development, which the proposal in this report explicitly tries to overcome.

In Figure 9 we report the simulation's throughput when varying the number of virtualized computing nodes in between 2 and 10. By the results, we can observe two different behaviours depending on the number of virtualized computing nodes used. Up to three computing nodes, the distributed-ECS solution presented in this report offers an improved performance with respect to the explicit message passing-based implementation. This is compliant with the original results in [5]. This result is mainly related to the fact that when running simulation events, the robots access a non-minimal simulation state both in read and write mode. At the same time, the probability that the accessed simulation state is on the same computing node is high (around 50%). Therefore, the likelihood that memory accesses can be actuated in place without the need for any actual page transfer is high. This is a scenario which definitely pays off, with respect to non-minimal data structures marshalling/unmarshalling—the data associated with the map being explored occupies around 3 pages in the simulation model.

When the number of distributed computing nodes increases, the overhead introduced by the transparent facilities offered by the presented architecture increases up to 30%. This behaviour is related to the fact that the degree of parallelism in such configurations increases—indeed, we have kept fixed the number of LPs while increasing the number of computing resources. Therefore, this can be regarded as a worst-case scenario for the ECS support. Indeed, the distributed protocol described in Section 4.4 blocks a given LP, due to cross-state synchronization, for a non-minimal amount of wall-clock time. This is related to the fact that multiple control messages are exchanged in order to synchronize the Local Clock of the LPs, to transfer the accessed memory pages, and to write back dirty pages. Since in the model a reduced number of LPs are involved in the synchronization, the likelihood that other LPs deliver a straggler message to synchronizing LPs gets higher. In this scenario, the probability that the transparent protocol undergoes a rollback operation is higher. This is reflected in the results, since the execution of a single ECS synchronization is retried multiple times. Anyhow, despite the adverse scenario, the protocol is able to keep the overhead quite reduced.

To show the effectiveness of the approach, in Figure 10 we report the speedup of the execution over a purely sequential execution of the same model, executed by relying on an efficient Calendar Queue [21]. By the results, we can observe that the offered speedup is non-minimal, showing that the whole experimental assessment has been carried out comparing competitive parallel runs.

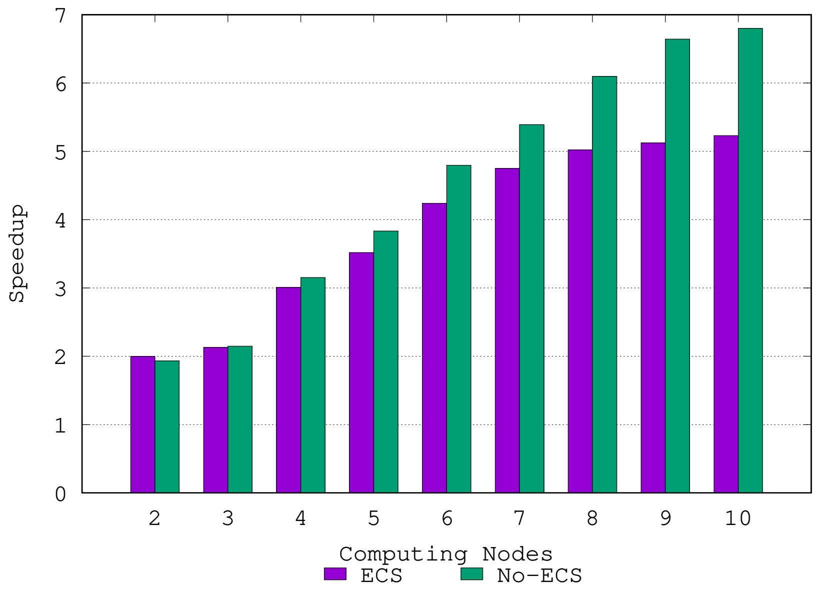

Figure 10: Speedup with respect to Sequential Simulation

6 Conclusion

In this report we have presented an architecture which allows LPs to rely on sequential-style memory accesses to read/write the simulation state of any LP in a distributed simulation environment. This approach has the benefit to significantly simplify the programming model according to which simulation models are implemented, and to enable a transparent deploy on generic distributed environments, such as clusters of (virtualized) computing nodes. Additionally, the preliminary experimental assessment which we have carried out shows that our solution is as well viable from a performance point of view, since the introduced overhead is almost negligible.

References

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

[2] S. a. McKee, "Reflections on the memory wall," *Proceedings of the first conference on computing frontiers on Computing frontiers - CF'04*, p. 162, 2004.

[3] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[4] A. Pellegrini, C. Montañola-Sales, F. Quaglia, and J. Casanovas-Garcia, "Programming Agent-Based Demographic Models with Cross-State and Message-Exchange Dependencies: A Study with Speculative PDES and Automatic Load-Sharing," in *Proceedings of the 2016 Winter Simulation Conference*, ser. WSC. IEEE Computer Society, Dec. 2016.

[5] A. Pellegrini and F. Quaglia, "Transparent multi-core speculative parallelization of DES models with event and cross-state dependencies," in *Proceedings of the 2014 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. PADS. ACM, May 2014, pp. 105–116.

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

[7] A. Pellegrini and F. Quaglia, "The ROme OpTimistic Simulator: A tutorial (invited tutorial)," in *Proceedings of the 1st Workshop on Parallel and Distributed Agent-Based Simulations*, ser. PADABS. LNCS, Springer-Verlag, Aug. 2013.

[8] D. Bruce, "The treatment of state in optimistic systems," *ACM SIGSIM Simulation Digest*, vol. 25, no. 1, pp. 40–49, jul 1995.

[9] H. Mehl and S. Hammes, "How to integrate shared variables in distributed simulation," *SIGSIM Simulation Digest*, vol. 25, no. 2, pp. 14–41, 1995.

[10] A. Fabbri and L. Donatiello, "SQTW: a mechanism for state-dependent parallel simulation. Description and experimental study," in *Proceedings of the Workshop on Parallel and Distributed Simulation*, 1997, pp. 82–89.

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

[12] L.-l. Chen, Y.-s. Lu, Y.-P. Yao, S.-l. Peng, and L.-d. Wu, "A Well-Balanced Time Warp System on Multi-Core Environments," in *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, ser. PADS. IEEE Computer Society, 2011, pp. 1–9.

[13] A. Pellegrini, R. Vitali, S. Peluso, and F. Quaglia, "Transparent and efficient shared-state management for optimistic simulations on multi-core machines," in *Proceedings 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS. IEEE Computer Society, Aug. 2012, pp. 134–141.

[14] B. P. Gan, M. Y. H. Low, J. Wei, X. Wang, S. J. Turner, and W. Cai, "Synchronization and management of shared state in HLA-based distributed simulation," in *Proceedings of the Winter Simulation Conference*, 2003, pp. 847–854.

[15] M. Y. H. Low, B. P. Gan, J. Wei, X. Wang, S. J. Turner, and W. Cai, "Shared State Synchronization for HLA-Based Distributed Simulation," *Simulation*, vol. 82, no. 8, pp. 511–521, 2006.

[16] R. Vitali, A. Pellegrini, and F. Quaglia, "Towards symmetric multi-threaded optimistic simulation kernels," in *Proceedings of the 26th International Workshop on Principles of Advanced and Distributed Simulation*, ser. PADS. IEEE Computer Society, Aug. 2012, pp. 211–220.

[17] A. Pellegrini and F. Quaglia, "A fine-grain time-sharing time warp system," *ACM Transactions on Modeling and Computer Simulation*, vol. 27, no. 2, May 2017.

[18] S. Peluso, D. Didona, and F. Quaglia, "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, 2011, pp. 169–177.

[19] A. Pellegrini, R. Vitali, and F. Quaglia, "Autonomic state management for optimistic simulation platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 6, pp. 1560–1569, Jun. 2015.

[20] D. Fox, J. Ko, K. Konolige, B. Limketkai, D. Schulz, and B. Stewart, "Distributed Multirobot Exploration and Mapping," *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1325–1339, 2006.

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