Transparent Multi-Core Speculative Parallelization of DES Models with Event and Cross-State Dependencies

Alessandro Pellegrini
pellegrini@dis.uniroma1.it

Francesco Quaglia
quaglia@dis.uniroma1.it

DIAG – Sapienza, University of Rome
Via Ariosto 25, 00185 Rome, Italy

ABSTRACT

In this article we tackle transparent parallelization of Discrete Event Simulation (DES) models to be run on top of multi-core machines according to speculative schemes. The innovation in our proposal lies in that we consider a more general programming and execution model, compared to the one targeted by state of the art PDES platforms, where the boundaries of the state portion accessible while processing an event at a specific simulation object do not limit access to the actual object state, or to shared global variables. Rather, the simulation object is allowed to access (and alter) the state of any other object, thus causing what we term *cross-state* dependency. We note that this model exactly complies with typical (easy to manage) sequential-style DES programming, where a (dynamically-allocated) state portion of object A can be accessed by object B in either read or write mode (or both) by, e.g., passing a pointer to B as the payload of a scheduled simulation event. However, while read/write memory accesses performed in the sequential run are always guaranteed to observe (and to give rise to) a consistent snapshot of the state of the simulation model, consistency is not automatically guaranteed in case of parallelization and concurrent execution of the simulation objects with cross-state dependencies. We cope with such a consistency issue, and its application-transparent support, in the context of parallel and optimistic executions. This is achieved by introducing an advanced memory management architecture, able to efficiently detect read/write accesses by concurrent objects to whichever object state in an application transparent manner, together with advanced synchronization mechanisms providing the advantage of exploiting parallelism in the underlying multi-core architecture while transparently handling both cross-state and traditional event-based dependencies. Our proposal targets Linux and has been integrated with the ROOT-Sim open source optimistic simulation platform, although its design principles, and most parts of the developed software, are of general relevance.

SIGSIM-PADS'14, May 18–21, 2014, Denver, CO, USA.

http://dx.doi.org/10.1145/2601381.2601398.

Categories and Subject Descriptors

I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete Event, Parallel*

Keywords

PDES; Parallelism Transparency; Speculative Processing

1. INTRODUCTION

Traditionally, Parallel Discrete Event Simulation (PDES) has been based on explicitly partitioning the entire simulation model into distinct simulation objects (also referred to as Logical Processes) [7] to be dispatched concurrently, whose states are disjoint and whose memory access operations (upon event processing) are confined within the state of the simulation object executing the event. This approach implicitly requires the application programmers to shift from a sequential programming model where the application is designed and coded to run serially (namely to process one event at a time) and to have the possibility to access any valid memory location upon the execution of whichever event. In other words, parallelism is achieved by a-priori, namely at code design/development time, forcing separation of the accesses to slices of the simulation model state, each one representing an individual object.

Undoubtedly, such a classical PDES approach has been the only way to parallelism at the time when (massively) parallel architectures were mostly based on clusters of (single-core) machines. On the other hand, the advent (and the large diffusion) of shared-memory parallel machines, such as multi-core and SMP machines, offered the technical possibility to directly share state information across different objects, e.g., by relying on a unique address space and/or operating system supported shared memory. However, the synchronization approaches provided by the PDES literature, and the actual simulation platforms relying on them, definitely need to be adapted in order to support correct concurrent execution of simulation objects while jointly allowing the possibility to directly share data and masking synchronization (and hence actual parallelization) to the application programmer. We note that the final target along the path of supporting shared data across different simulation objects actually translated into enabling a sequential-style programming approach (augmented with the concept of "object", which allows for improving expressiveness while coding complex simulation models), characterized by full access capabilities to any valid memory location (logically belonging to the state of any involved object) upon executing whichever event, in either read or write mode.

One proposal along this direction has been recently provided in [19] for the case of simulation code based on the C programming language, where direct sharing of information across the simulation objects has been supported for the case of global variables included within the simulation program. This approach has been based on multi-versioning plus code instrumentation (for accessing multi-version chains) as the means to achieve programmer transparent concurrent accesses to global variables jointly to optimistic synchronization [12]. However, sharing based exclusively on global variables limits the actual possibility to share data in size, given that the storage for global variables is statically defined at compile time. Also, it still constraints the programmer, who is not allowed to directly access arbitrary slices of (dynamically-allocated) memory destined to keep portions of the simulation model (e.g. by having them logically representing the state of a generic simulation object).

In this article we exactly tackle the above problem, namely how to support direct access by concurrent simulation objects to memory locations that are dynamically allocated by any simulation object, and logically included within its local state via, e.g., pointer based referencing. The same pointers can be used as payloads of events so that the recipient simulation object can use them to directly access the local state of a different object, in either read or write mode. This breaks disjointness in memory access at the programming level, hence enabling the support for sequential-style DES programming (where any valid memory location keeping a portion of the state of whichever object, is accessible while processing any simulation event), and creates a new kind of dependency that we term *cross-state* dependency, which stands as complementary with respect to the classical event dependency proper of PDES. On the other hand, guaranteeing correct (e.g. causally consistent) execution of simulation events in the presence of cross-state dependency across concurrent simulation objects requires proper application-transparent synchronization mechanisms to be put in place, which we provide in this article.

Overall, the contributions by this article can be summarized as follows:

1) We present the design and implementation of an innovative memory management architecture, oriented to Linux systems, which allows to detect the materialization of cross-state dependencies across simulation objects that are run concurrently, in an application-transparent manner. The architecture requires a minimal patch to the Linux kernel, given that it is almost exclusively based on an external loadable module.

2) We present a synchronization scheme, entailing speculative processing, which takes into account both event and cross-state dependencies and allows the parallel run to mimic a classical sequential one where the simulation events are processed in non-decreasing timestamp order, while jointly being allowed to access any valid memory location belonging to the state of the simulation model (namely any memory location logically belonging to the state of some object). On the other hand, our scheme, which we name ECS (Event and Cross-State synchronization), allows running simulation events destined to different simulation objects concurrently (again transparently to the programmer).

All the presented software modules and algorithms have been integrated within the ROOT-Sim open source simulation package [10] and are available for download[1]. Further, experimental data for an assessment of the whole ECS architecture are provided.

The remainder of this article is organized as follows. In Section 2, we discuss literature results related to our proposal. The innovative memory management architecture transparently tracking cross-state dependencies and the actual ECS synchronization protocol are presented in Section 3. The results of the experimental study are provided in Section 4.

2. RELATED WORK

The issue of bypassing state disjointness for concurrent objects in PDES systems has been dealt with by several studies. The work in [2] discusses how state sharing might be emulated by using a separate simulation object hosting the shared data and acting as a centralized server. This proposal also introduces the notion of *version records*, where multi-versioning is used for shared data in order to cope with read/write operations occurring at different logical times, and to avoid unneeded rollbacks of the centralized server in case of optimistic synchronization. This is an approach similar to the one proposed in [17], where a theoretical presentation of algorithms to implement a Distributed Shared Memory mechanism is provided in terms of protocols to keep replicated instances of a variable coherent. In particular, one of the provided algorithms proposes to implement variables as multi-version lists where write operations install new version nodes and read operations find the most suitable version. The above approaches are different from what we propose given that read/write access to shared variables is mapped to message-passing (namely, event schedule operations), while we support a truly sequential-style access to any (by default sharable) buffer within the simulation object states, e.g., via pointers. Also, in our proposal sharing is not limited to a particular memory slice (such as the state image of the centralized server), while we allow access, and hence sharing, of any memory buffer representing a portion of the whole simulation model state. Also, by design the above approaches are strongly oriented to distributed simulation environments, while we target the trend of shared-memory/multi-core machines.

In another proposal [6] the notion of *state query* is introduced. A simulation object 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 so as to invalidate the query. Again, this approach relies on message passing, and is not transparent to the application programmer.

The work in [9] proposes to integrate the support for shared state in terms of global variables, by basing the architecture on [3]. Although this proposal supports in-place read/write operations as we do (i.e., simulation objects directly access the only copy of the data, avoiding a commit phase at the end of the execution of an event), it provides no transparency, as the application-level code must explicitly register a simulation object as a reader/writer on shared variables. Our proposal avoids this limitation by also allowing the sharing of dynamically-allocated buffers, for which pre-declaration of the potential need to access cannot be raised at startup (hence intrinsically leading actual access

[1] http://svn.dis.uniroma1.it/svn/hpdcs/root_sim/trunk

to be determined as a function of the specific execution trajectory while running the application). The issue of transparency has been tackled in [19], where shared data are allowed to be accessed by concurrent objects without the need for pre-declaring the intention to access. This has been achieved via user transparent software instrumentation, in combination with a multi-version scheme, either allowing the redirection of read operations to the correct version of the data (on the basis of the timestamp) or forcing rollbacks of causally inconsistent reads. However, this solutions is limited to the management of global variables, while our proposal is suited for allowing data sharing across dynamically allocated memory chunks logically incorporated within the state of each individual simulation object, while providing parallelism and synchronization transparency.

In the context of the High-Level-Architecture (HLA), proposals for supporting shared-state can be found in [8, 15]. They are again targeted at distributed environments, since they are based on a middleware component which relies on a timestamp-ordering approach for implementing a request/reply protocol. Additionally, these approaches are targeted at the conservative synchronization protocol, where there is no need to detect and handle causality violations, while we target optimistic synchronization.

The work in [4] proposes a framework targeted at multi-core machines and based on Time Warp, where so called Extended Logical Processes (Ex-LP), defined as a collection of LPs, have public attributes that are associated with variables which can be accessed by LPs in other Ex-LPs. The work proposes to handle shared attributes accesses by relying on a specifically targeted Transactional Memory (TM) implementation, where events are mapped to transactions and the actual implementation of the TM is based on [9]. One core difference between our proposal and the one in [4] is that the latter requires a-priori knowledge of the attributes to be shared, which need therefore to be a-priori mapped to TM managed memory locations. Rather, our proposal allows for sharing any memory area, without the need for a-priori knowledge of whether some sharing on a specific area can occur. This increases the level of transparency, again allowing a truly sequential-style programming model to be exposed to the programmer. In fact, she is allowed to let any simulation object that takes control touch any valid memory location within the global simulation state without the need for any particular care, just like it occurs in sequential-style programming and related sequential execution scenarios. 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.

3. EVENT AND CROSS-STATE SYNCHRONIZATION

3.1 Cross-State Dependency Tracking

In this section we present the memory management architecture we have designed and developed in order to support cross-state dependency, and to actually track the materialization of such type of dependency across simulation objects that are run concurrently. Let us stress again that our architecture supports cross-state dependency in a fully transparent manner with respect to the application level software. As an additional preliminary note, in our design we targeted

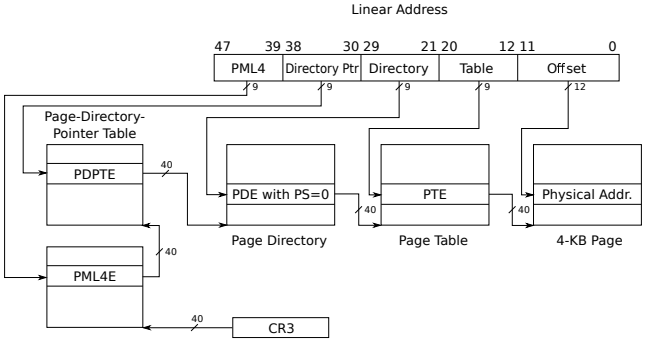

Figure 1: The paging scheme in x86_64 processors.

PDES platforms relying on the multi-threading paradigm. These have been shown to provide a set of benefits and to support optimized resource usage policies (see, e.g., [4, 11, 27, 26]) when compared to the traditional counterpart where parallelization is achieved by running a set of single-threaded processes within the simulation platform. Overall, we designed a memory management architecture allowing not to loose the benefits from multi-threading.

On the basis of the above considerations, we target the scenario where multiple threads can take care of dispatching whichever simulation object for execution (although we will still rely on temporary-binding schemes between objects and worker threads in order to cope with, e.g., locality and other performance-related aspects), which takes place by simply calling an event-handler function, with proper input parameters, along that thread. Also, we target the `C` programming language, so that the event-handler function taking control at the application level is an ANSI-C function. As typical for DES-style coding rules, this function has a set of input parameters which includes the state base pointer, namely the memory address of the data structure starting from which the object is allowed to access any other dynamically allocated buffer belonging to its state via pointers.

In our architecture, virtual memory is destined for usage to any simulation object according to *stocks*. More in detail, when the object requests new memory buffers (which we support via the traditional `malloc` service, redirected to a proper memory allocator), the memory management architecture reserves an interval of page-aligned virtual memory addresses, namely the stock, which is achieved via the standard `mmap` POSIX API. We note that any page in the stock is an empty-zero page, thus being not really allocated in memory until the first read/write access to it is performed. This is the standard management performed by POSIX (e.g Linux) systems.

To understand how we use the stock for supporting cross-state dependency tracking, let us consider the actual paging scheme offered by x86_64 architectures. As shown in Figure 1, any 64-bit logical address has only 48 valid bits, which are used as access keys for a 4-level paging scheme, ultimately supporting pages of 4KB in size. The top level page table is called PML4 (or also PGD—Page General Directory) and keeps 512 entries. All the other page tables, operating at lower levels, also have 512 entries each. In our design, the stock of virtual memory pages destined for allocation of memory buffers for a given simulation object corresponds to the set of contiguous virtual-pages whose virtual-to-physical memory translation is associated with a single entry of the

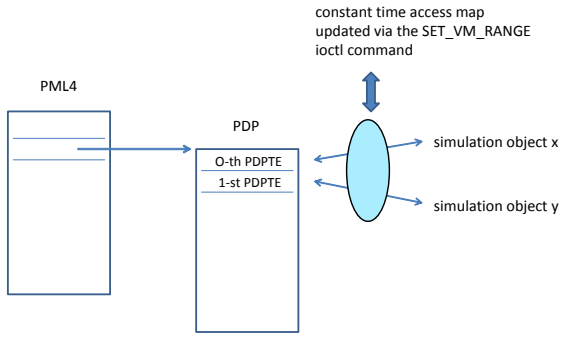

Figure 2: Example of association between stocks of virtual memory pages and simulation objects.

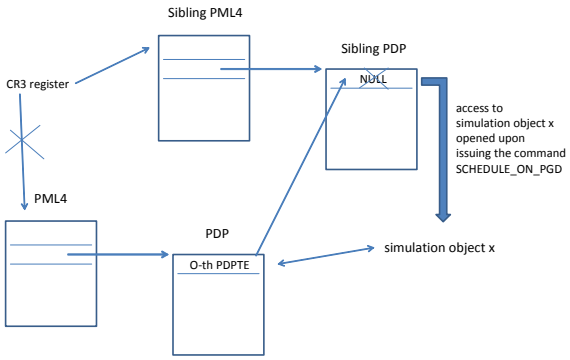

Figure 3: Example scenario where the memory stock associated with simulation object x is opened for access onto a sibling PDP page table.

second-level page table, which is called PDP—Page Directory Pointer (its entries are therefore referred to as PDPTE). Note that a single stock corresponds to 512^2 pages, for a total of 1GB of virtual memory. Hence, a single stock allows managing an object-state requesting up to 1GB of (dynamic) memory. On the other hand, reserving multiple stocks for a same simulation object will lead to manage object states reaching multiple gigabytes in size.

We have created a special device file (whose driver is loaded into the Linux kernel via an external module) which can be handled via proper `ioctl` commands, whose logic we have implemented within the driver. The `SET_VM_RANGE` command allows the special device to register the stocks to be reserved, and their association to the simulation objects (which are distinguished via classical unique numerical identifiers, as in typical PDES-platform implementations). When this command is issued, the state of the device file changes so that the driver sets up a kernel-level map (accessible in constant time) where for each reserved stock, which is logically related to one entry of a PDP page-table, the identifier of the simulation object destined to use that stock is recorded. In Figure 2 we show an example where a given PDP table has its 0-th entry, and hence the corresponding stock of virtual memory pages, reserved for object x, and its 1-st entry reserved for object y.

By this kind of organization, if simulation object x accesses any virtual address included in the stock reserved for object y, we know that such a memory access (which can be either in read or write mode in our execution model) is occurring outside the boundaries of its local state, and is actually involving the state of another object. Therefore, we are experiencing a cross-state dependency. We recall again that this may occur, e.g, if object y scheduled a simulation event destined to object x, carrying as payload the pointer to some memory buffer belonging to the state of y, just to indicate to x where to take (and possibly update) the information requested for processing the event.

The core problem to cope with in order to exploit the stocks as the means to capture whether the generic simulation object x (currently dispatched for execution along any worker tread WT_i within the PDES platform) is materializing a cross-state dependency is related to how to determine that event processing gives rise to a memory reference falling outside the boundaries of the stocks currently reserved for object x. We note that classical memory protection mechanisms supported by the operating system (and related segmentation-fault handling schemes) are not suited for our purposes. Particularly, given that we are targeting multi-threaded PDES platforms, we cannot simply a-priori protect the accesses to stocks that are reserved for simulation objects other than x upon dispatching x along any worker thread WT_i. This is because these simulation objects might be requested to run concurrently with respect to x along other worker threads, which all share the same page table and experience the same protection rule as WT_i. Overall, closing to WT_i the access to the stocks not reserved for simulation object x upon dispatching it (e.g. via the `mprotect` POSIX API) would lead to a change of the state of the page table where any other thread would not be allowed to access those stocks. This would clearly hamper concurrency, also leading to unneeded memory faults (by threads running objects other than x) in contexts where the object x does not require any access to "remote" stocks while processing the event. On the other hand, addressing the above problem via (user transparent) code instrumentation would require to instrument not only memory write instructions (as typically done when supporting transparent incremental checkpointing in optimistic PDES platforms [28, 20]), but also all the memory reads, which would lead to overhead to be paid even in case no cross-state dependency, although admissible, will ever materialize (a scenario leading the tracking of memory read operations un-useful on the side of synchronization tasks related to cross-sate dependencies).

In order to cope with the above depicted core issue, we have devised a memory management architecture where any worker thread WT_i is associated with a sibling PML4 page table, whose entries point to sibling PDP page tables. The sibling page tables (both PML4 and PDP) destined for usage by a worker thread can be instantiated by relying on the `GET_PGD` command included in the special device file driver, which returns a descriptor for subsequent operations. By default, the entries of the sibling PDP page tables, which are associated with the stocks that have been destined for usage by the simulation objects, are all set to NULL. This means that they do not allow to reach the lower level page tables, hence not allowing access to any already allocated stock (therefore, any attempt to access the stocks will lead to a memory fault). On the other hand, when WT_i dispatches simulation object x for event execution, the entries of the PDP sibling tables that correspond to the virtual memory stocks destined for usage by x are "opened" to correctly allow the retrieval of the lower level page tables that contain the actual mapping of virtual-to-physical memory

(or indications about whether the pages are not present, e.g., they are swapped-out pages). This is done by copying the corresponding entries of the original PDP tables onto the destination entries within the sibling PDP page tables (see Figure 3 for an example scenario where the stock associated with simulation object x is again related to the 0-th entry of a given PDP page table). In our architecture, this operation can be executed via the additional `SCHEDULE_ON_PGD` command we have included within the special device file driver, which can be issued via the `ioctl` interface. In other words, via this command, the worker thread is allowed to switch to what we refer to as *simulation-object mode*, where the only accessible stock is the one associated with the dispatched object (say x in the example discussion), while the other stocks are no way accessible (given that their corresponding entries into the sibling PDP page tables are still set to NULL). As schematized in Figure 3, in our implementation this operation also leads to a change of the `CR3` register (namely, the page table pointer register in x86_64 processors), thus allowing to switch to the sibling PML4 for virtual-to-physical address resolution purposes.

Having different sibling PML4 tables, associated with the different concurrent worker threads, leads to the possibility to concurrently dispatch and execute different simulation objets (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, our architecture entails a stock allocation policy that beyond calling `mmap`, also explicitly writes a null byte into one single virtual page of the stock (the initial one). 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), which guarantees the existence of the PDP entry associated with the stock, to be filled into the corresponding sibling PDP entry upon dispatching the object owning the stock.

Two additional points need to be discussed. First, having all the stocks closed for access by the worker thread, except the one(s) related to the dispatched object, leads (as noted before) to memory faults in case of a memory access to stocks other than open one(s), namely in case of materialization of a cross-state dependency across concurrent simulation objects. However, these faults cannot be tracked (and handled) via classical segmentation-fault handling given that the "remote" stocks have already been validated via `mmap`, and the Linux kernel would simply lead the fault to reallocate the whole chain of page table entries for mapping the accessed virtual page in memory. This would lead the whole system to a state where for the same virtual page we would have multiple chains of page table entries representing its state (e.g. the frame used for mapping the page, which might be different along the multiple chains of page table entries) which is a discrepancy not directly hand-able by the Linux kernel (except if using invasive patches). To avoid this scenario, upon installing the driver for the special device file, via loading

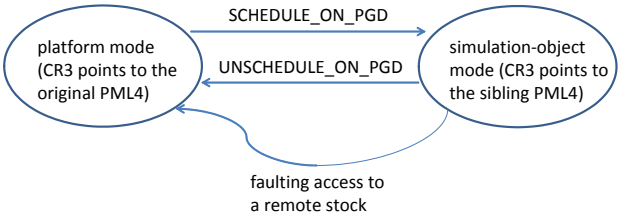

Figure 4: The state diagram for switch operations between original and sibling PML4 page tables.

the external module, we change the IDT table (directly accessible via the `IDT` register) in order to make the pointer to the page-fault handler point to an ECS-proper handler (rather than the original `do_page_fault` kernel function). In case the fault is not related to accesses to remote stocks within the sibling paging scheme, then the original handler is invoked. Otherwise, the ECS-handler pushes control back to user mode in order to let the PDES platform to actuate ECS synchronization policies, exactly aimed at coping with cross-state dependencies.

Upon a memory-fault occurring on sibling PDP entries, due to cross-state dependency materialization, the faulting thread is put back into what we call *platform mode*, which implies that it is switched back onto the original PML4. This is done in order for this thread to operate any memory access required to reconcile the execution of the concurrent objects according to ECS synchronization. This aspect will be treated in detail in the next section. On the other hand, in case event processing at the currently dispatched object ends, the worker thread can switch back to platform mode on demand (hence gaining access to any memory location or data structure supporting the parallel execution) by using the `UNSCHEDULE_ON_PGD` command that we have implemented within the driver, which can be triggered by again exploiting the `ioctl` POSIX API. In Figure 4 we show the state diagram where the events causing the switch between simulation-object and platform modes are depicted.

Second, our architecture needs anyway to co-exist with the kernel scheduler, which poses issues on the side of managing the sibling PML4. Particularly, all the threads within a same Linux process share the same memory management information (the so called memory context), including the pointer to the original page table. This pointer is used by the kernel scheduler upon re-dispatching the thread after it has been context-switched off the CPU. Particularly, this pointer is reloaded into the page-table pointer register `CR3` upon the occurrence of a context switch that gives control to the thread. However, if the thread was executing in simulation-object mode, `CR3` would need to be filled with the address of the sibling PML4 (rather than the original page table). To achieve this, a minimal patch to Linux has been adopted, which has been located right at the end of the kernel `schedule` function. The patch simply checks whether the value of a special function-pointer we inserted into the kernel is not null, in which case the function-pointer is invoked, which gives control to a proper `CR3` manager implemented within our external module. This manager checks whether the thread is running into simulation-object mode (which can be done by checking per thread meta-data that were setup via the `SCHEDULE_ON_PGD` command) and, in the positive case, it loads the sibling PML4 pointer into the `CR3`

register (thus maintaining the simulation-object mode when running the thread). Note that the aforementioned special pointer is exported as a kernel symbol, and can be set to a value different from NULL upon inserting the external module. If this pointer is not set the Linux kernel behaves as usual, by simply restoring the `CR3` register according to the standard rules when the thread is rescheduled after a context switch.

Before ending the presentation of the memory management architecture, let us discuss two aspects related to the actual memory allocation support for the application code and to safety of dual-mode execution (platform vs simulation-object) in the presence of third party libraries. As for the first aspect, we integrated the memory management architecture with the DyMeLoR open source allocator [25, 20], explicitly targeting memory allocation needs in optimistic PDES platforms. It intercepts dynamic memory calls by the simulation object (e.g. `malloc` calls) and handles them by managing (and delivering to the simulation object) memory chunks located into a pre-reserved memory segment. DyMeLoR also keeps bitmaps to determine the currently in use chunks and the dirty ones, which allows for taking (incremental) snapshots and restoring past states of the simulation object (while allowing dynamic memory to be used by the object). Integration of DyMeLoR with the currently presented architecture has been straightforward given that, rather than relying on actual `malloc` implementations for pre-reserving the segment destined to allocate the chunks for a given simulation object, in the integrated architecture we let DyMeLoR rely on the stock allocator. Hence, the virtual memory segment managed by DyMeLoR boils down to the stock of virtual memory pages supported in the presented architecture. We note that identifying dirty chunks in DyMeLoR relies on compile/link time instrumentation of memory write instructions within the application level code. The outcoming memory access-tracking scheme is completely different from the support we are offering for cross-state dependency tracking, which is able to intercept read access (not only write access) to whichever application destined memory area without the need for instrumenting all the read operations.

As for the second aspect raised above, DyMeLoR is shipped with wrappers for ANSI-C stateless libraries, so that any memory allocation by these libraries (such as `strdup`) is still handled by DyMeLoR according to the above depicted scheme. Also, the third-party library interfaces are redirected to an actual logic which is statically linked to the application code, hence not requiring intervention by the dynamic linker. This automatically avoids page table updates while running in simulation-object mode which would otherwise be caused on sibling page tables by memory mapping actions by the dynamic linker (in case the shared libraries were invoked by the application code while running in simulation-object mode). The only limitation in terms of library usage by the application code is that the whole architecture does not yet support stateful libraries (e.g. `strtok`)[2], whose data structures could be involved in read/write operations caused by library invocations by whichever concurrent simulation object, thus possibly creating indirect cross-state dependencies not catchable via the above described stock management policies. Coping with the employment of stateful libraries will be the target of future work.

As an additional note, our approach requires reloading the `CR3` register anytime we switch between platform and simulation-object mode. The penalty incurred consists in flushing the TLB right upon loading a new value into `CR3`, which is done automatically by the firmware logic of x86_64 processors (this is anyhow required in order to make the access-rule of the target page table – original vs sibling – visible after the switch, which cannot be achieved without refilling the TLB). However, the data cache does not require to be invalidated, hence we expect that the cost for TLB renewal would look affordable as soon as a certain level of locality is exhibited while running either in platform or in simulation-object mode. As for this aspect, the reliance on DyMeLoR would favor locality in simulation-object mode, given that DyMeLoR implements policies aimed at maximizing virtual-memory contiguousness of the memory chunks delivered for usage by the simulation object.

Finally, we note that the on-demand switch to simulation-object mode or (back) to platform mode requires invoking the `ioctl` system call. While the cost of system calls has been traditionally considered an issue in high performance computing, especially when dealing with fine grain tasks, such costs are nowadays definitely reduced thanks to `sysenter` and `sysexit` machine instructions, which are explicitly designed for low-latency system calls, by relying on operating systems with a flat memory model and no segmentation. These instructions have been optimized by reducing the number of checks and memory references that are normally made so that a call or return has been shown to take less than one-fourth the number of internal clock cycles when compared to the traditional approach based on the `int` instruction, which was explicitly based on segment-gate retrieval and segmented-to-linear memory addressing translation.

3.2 The Enhanced Synchronization Scheme

In this section we provide the core mechanisms underlying ECS synchronization. The main difference between classical event-based synchronization and ECS lies in that ECS-synchronization tasks not only aim at letting each simulation object process its events in non-decreasing timestamp order (in accordance with the reference correctness criterion for PDES platforms only entailing event dependencies). Rather, ECS synchronization also aims at allowing any cross-state dependency materialized at simulation time t to let the involved object (namely the one accessing remote stocks reserved for other objects) to observe the state snapshot that would have been observed at simulation time t in a sequential-run, where simulation events were processed in globally non-decreasing timestamp order across all the objects.

We base ECS synchronization on the following two innovations:

A) the introduction of temporary object blocking phases, which may even lead to temporary block the execution of an already dispatched object (namely of an already dispatched simulation event at that object);

B) the introduction of so called *rendez-vous* events, which are kinds of system level simulation events not causing

[2]One exception is clearly the `malloc` library, for which the above described ad-hoc architecture has been put in place. On the other hand, for `stdio` one can rely on, e.g., the I/O management subsystem presented in [1], explicitly targeting consistency of I/O operations in optimistic synchronization transparently to the application code.

updates on the destination object state, but only driving block and unblock actions for processing activities at the objects. These will be exploited to temporarily disable a simulation object to perform updates of its state along the simulation-time axis, given that its state snapshot is currently being involved in a cross-state dependency.

We note that point A leads to an event processing model where control (on any worker thread) can return to the platform layer before an already started event-processing phase actually ends. This takes place according to an interrupt-driven scheme, different in nature from event-preemption ones that have been put in place in optimistic PDES systems to squash the execution of events that are detected to be causally inconsistent while still being processed, for either performance or infinite-loop avoidance reasons [24, 18]. In fact, these proposals have been typically based on polling (see, e.g., [24]) to be explicitly actuated by the event processing code, which is used to periodically query the platform layer to check whether no straggler event/antievent was delivered. On the other hand, point B leads to bridge PDES execution models with Transactional Memory models, particularly by having read/write operations across different stocks serialized according to the logical time for their occurrence.

In our proposal, each simulation object x is associated with a cross-state dependency set we refer to as CSD_x, which records the identifiers of all the simulation objects towards which x has materialized a cross-state dependency during the processing of an event. CSD_x is initialized as empty upon dispatching object x for the execution of any new event, and gets possibly updated while processing the event. ECS synchronization exploits the ad-hoc memory-fault management architecture presented in the previous section in order to detect that simulation object x is accessing a remote memory stock, say the stock associated with object y, in either read or write mode, while processing its next event, say e_x. The identity of the object towards which the cross-state dependency is being materialized (say y in our example discussion) is also known, given that the ECS memory-fault handler, which pushes the thread back in platform mode, notifies such an identifier into the thread user-mode stack. The memory fault occurrence gives rise to the following algorithmic steps:

1. Execution of e_x is temporarily blocked, hence object x transits into a block state;

2. A rendez-vous unique identifier is generated and assigned to the event e_x, which we refer to as $rvid(e_x)$.

3. A special rendez-vous event e_y^{rv} is scheduled for object y, marked with timestamp equal to the timestamp of event e_x, and with its identifier (formally $ts(e_y^{rv}) = ts(e_x)$ and $rvid(e_y^{rv}) = rvid(e_x)$). We note that rendez-vous events are not generated by the application layer, rather they are platform-generated events. Hence they do not have any associated processing rule at the application level.

Rendez-vous events are incorporated into the event list of the destination object as if they were traditional events. Given that we are targeting optimistic synchronization, this means that a rendez-vous event may be a straggler event (in case its timestamp is lower than the timestamp of some already processed event at the destination). They need therefore to be processed along the sequence of events of the destination, and the processing actions are platform-level actions proper of ECS (given that, as hinted above, no application level processing rule is – and needs to be – specified for rendez-vous events).

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

1. Object y is put into a block state;

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

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

1. It inserts the identifier of the sender object, namely y, into CSD_x.

2. It puts the simulation object x back in the ready state (so that it can be eventually re-dispatched along some worker thread, thus resuming the execution of the originally interrupted event e_x).

At this point we know that simulation object y is blocked (thus not being currently allowed to process its events), hence the snapshot of its state is available to simulation object x for read/write operations, such as the operation that originally gave rise to the ECS memory fault and to the cross-state dependency being handled via the rendez-vous. However, upon re-dispatching object x (which leads to resuming the processing of e_x), the involved worker thread cannot transit into simulation-object mode by only opening the stock(s) associated with x into the sibling page tables. Rather, we also need to open access to the stock(s) associated with object y. In our architectural support, this can be still achieved via the `SCHEDULE_ON_PGD` command, given that this command has been augmented with capabilities to acquire a set of identifiers whose stocks need to be opened within the sibling page tables when the worker thread transits into simulation-object mode. Particularly, upon re-dispatching object x, the `SCHEDULE_ON_PGD` command is issued with in input the set $x \cup CSD_x$, which for our example discussion, contains the identifiers of both the objects x and y.

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

We only need to discuss how the finalization of the processing phase of e_x is handled. Essentially, we need to generate notifications that the stocks associated with simulation objects towards which cross-state dependencies have been

materialized are no longer locked for access by object x. Hence, the owner simulation objects can resume their normal processing activities (thus they can resume from the block state). This is achieved via the following steps executed right after the processing of event e_x at object x:

1. An unblock-event e_k^{ub} is sent towards any object k whose identifier is logged within CSD_x. These events are marked again with no timestamp value, but with the rendez-vous identifier of the event e_x originating the cross-state dependency. Then CSD_x is reset as empty.

2. Upon the delivery of e_k^{ub}, the recipient simulation object is simply put back as ready for being dispatched (hence exiting the block state).

However, some additional mechanisms are required in order for ECS to provide correctness and to also ensure progress of the parallel run. These two aspects will be dealt with in the following subsections.

3.2.1 Correctness

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

In order to handle such mutual dependency, we devise the following scheme. When the event e_x is rolled back, we simply send an anti-event for the rendez-vous event e_y^{rv} that was scheduled while processing e_x. Given that e_y^{rv} was actually incorporated into the event list of the destination object y, the arrival of the anti-event gives rise to a classical annihilation that possibly rolls back y to the latest processed event with timestamp less than $ts(e_y^{rv})$. This solves the problem of rolling back object y due to the rollback of a rendez-vous generating event e_x on simulation object x.

On the other hand, in case the rollback is originated on object y, and pushes this object to a simulation time less than $ts(e_y^{rv})$ (which leads to undo the execution of e_y^{rv}), the following actions are taken by ECS. A special rendez-vous-restart event e_x^{rvr}, marked with the original rendez-vous identifier (namely $rvid(e_x)$) is sent out towards object x. This special event has the aim of annihilating the processing of the original instance (while not removing it from the input queue), which will lead to ultimately undo e_y^{rv} via an anti-event. Given that when processed after the rollback, the event e_x will give rise to a rendez-vous marked with a different identifier (with respect to the rolled-back rendez-vous instance), no mismatch will occur in any annihilation phase for rendez-vous events associated with different incarnations of their generating event (which also avoids cycles in the annihilation process).

Also, all the other types of events used in ECS, such as acknowledgment and unblock events, are not actually incorporated into the event lists of the simulation objects, thus being inherently ephemeral, and not requiring particular care in the rollback scheme. Assuming FIFO communication across the objects, these events can be simply discarded at the recipient side if the rendez-vous associated with their corresponding identifier (e.g. $rvid(e_x)$ in case of the acknowledgement event sent to x upon the rendez-vous) is no more in place.

3.2.2 Progress

A bit more complex to deal with in ECS is the guarantee of progress. Specifically, care must be taken to avoid deadlocks and live-locks, and the domino-effect in the rollback scheme. Let us first consider the dealock/live-lock issue. A deadlock may arise in case of rendez-vous events cyclically involving a set of simulation objects, where the rendez-vous associated with the minimal timestamp along the cycle leads the simulation object raising this rendez-vous to wait for the rollback of a different object that is, in its turn, in the block state due to a different rendez-vous it issued, which needs to be completed. An example situation of this type is shown in Figure 5, where object x issues at time t_1 a rendez-vous towards object z, which is waiting for object y to reach time t_3 for a rendez-vous between z and y. On the other hand, object y is waiting for x to reach time t_2 for a rendez-vous with it. To avoid deadlock scenarios, we can simply adopt the rule that, in case a rollback needs to be executed by a simulation object x which is currently blocked due to a rendez-vous it generated while processing and event e_x, this object is simply resumed from the block state by also squashing the finalization of the rendez-vous (this will lead to manage the rollback of the rendez-vous as explained above, e.g., by issuing the anti-event for the already sent out rendez-vous event). We note that this implies that the current stack seen by the object also needs to be refilled with correct information (since, upon resuming, its context will no more be the processing context for the rendez-vous generating event). We note that this is a problem similar to the one of restoring the correct stack of the object upon resuming the processing of a rendez-vous generating event that lead it into the block state (so that the worker thread currently executing this object passes control to a different object, which needs to operate on a proper stack image). Details on how we handled this issue in our implementation, where the cross-state dependency tracking architecture and ECS have been integrated into the ROOT-Sim open source optimistic simulation platform, will be discussed in Section 4. We also note that annihilating the rendez-vous event via the corresponding anti-event is safe even in case the destination object is currently blocked waiting for the finalization of the rendez-vous. In fact, it can be simply resumed from the block state (again with proper stack image manipulation) and can be rolled back, thus possibly altering its state image safely given that the image does no more need to be locked for the access by a different object in a rendez-vous.

We note however that unblocking the object generating a rendez-vous so as to prevent deadlock in case a rollback is

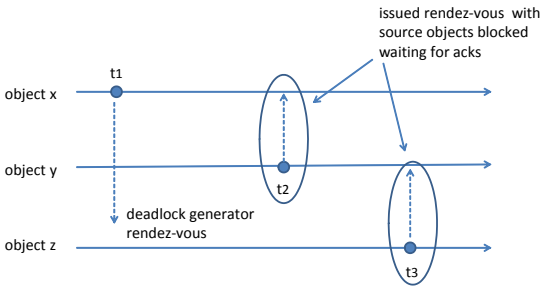

Figure 5: Example scenario with deadlock originated by a rendez-vous generating event at time t_1 processed by simulation object x.

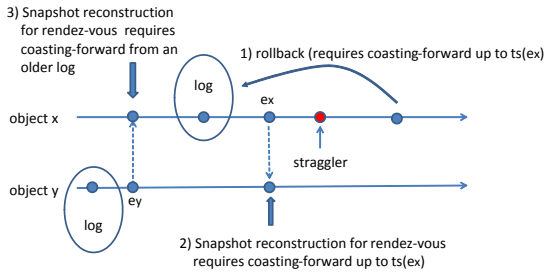

Figure 6: Example scenario with domino-effect due to a rollback originated on simulation object x.

required may, in its turn lead to live-lock. Specifically, live-lock may in principle arise in case of simultaneous events materializing circular cross-state dependencies across multiple simulation objects. Each object x along the circle, executing an event e_x at simulation time $ts(e_x)$, is hit by another object due to a cross-state memory faulting access at the same simulation time, which may lead to request the rollback of the events generating the rendez-vous circularly. This is known to possibly lead the rollback circle to reappear indefinitely [14]. To overcome this problem, we need a priority management scheme for simultaneous events, that needs to be reflected also on the management of rendez-vous events. Particularly, if we have two events e_x and e_y such that $ts(e_x) = ts(e_y)$, and we have a priority scheme telling that $e_x \rightarrow e_y$ (namely, e_y is identified as causally dependent on e_x), then we need to enforce that any rendez-vous event e_y^{rv} generated by e_x is also causally related to e_y according to $e_y^{rv} \rightarrow e_y$. This way, the rendez-vous occurrences that are caused by events having the same timestmap are anyway sequentialized according to the priority scheme. We note however, that the guarantee of progress in (optimistic) PDES systems in the presence of simultaneous events is a more general problem, with respect to what we might experience in ECS, and has been extensively studied in literature [13]. Hence different literature solutions for tie-breaking simultaneous events (see, e.g., [16]) can be exploited for integration with ECS according to the scheme suggested above.

The final issue to cope with is the domino-effect in the rollback scheme. Particularly, by the ample literature on log/restore in optimistic PDES systems (see, e.g., [21, 22, 23]), we know that sparse state saving, which avoids logging the simulation object state after the processing of each event, allows for optimizing the performance tradeoff between logging cost and restore cost. However, state restore at time t requires the simulation object to be rolled back to the latest state log with time less than or equal to t, and then to fictitiously reprocess intermediate events up to t in a silent mode (namely with no interactions with other objects), which is also known to as coasting-forward. In ECS this is no longer possible since a coasting-forward event might be a rendez-vous generating event. Hence, in order for this to be re-processed, the simulation object originally hit by the rendez-vous also needs to rollback at the time of the rendez-vous, so as to provide its state snapshot for correct access by the object performing coasting-forward. It is easy to show that this may lead the originally rolling-back object to rollback further back along simulation time, according to the domino-effect. An example is shown in Figure 6, where in order to execute the coasting-forward involving event e_x at object x, we need to reconstruct the snapshot of object y at time $ts(e_x)$, But this leads to the need for processing e_y in a coasting-forward, which in turn leads x to restore its state to a time less than $ts(e_y)$. To avoid the need for executing coasting-forwards leading to rollback interactions with other objects (thus avoiding the domino-effect), our approach is based on complementing the selected sparse state saving algorithm by forcing the log of the state of a simulation object right after the processing of a rendez-vous generating event. This will lead to the scenario where no rendez-vous generating event will ever be included in the sequence of events between to subsequent logs of the same simulation objects. Hence no rendez-vous generating event will need to be reprocessed in any coasting-forward phase. On the other hand, a rendez-vous generated event also needs to be excluded by any coasting-forward, since for these events the rendez-vous source object may have performed updates into the state of the target object. To avoid a rendez-vous generated event to be included in any coasting-forward phase, we can again force a state log of the involved object right after the event is processed.

4. EXPERIMENTAL STUDY

4.1 Test-bed Platform

We integrated ECS within the open source ROOT-Sim package [10], particularly the symmetric multi-threaded version presented in [27]. A few relevant modifications to this simulation platform have anyhow been made for integration purposes. Most relevantly, we have created stack-separation across the different simulation objects, by locating the stack of each object in the initial part of a stock of memory destined for object usage. Also, execution resume in the different stacks, by also providing the correct processor and stack image, has been supported via `setjump` and `longjump` POSIX APIs. These have also been used as the support for, e.g. squashing the stack image in case a rollback occurs while the simulation object is in the block state (which eventually leads the object to resume execution with a different context). As for the (temporary) binding of simulation objects to threads, we still relied on the already supported policies. Also, the simulation objects currently bound to a given worker thread are still dispatched according to the Lowest-Timestamp-First policy. However, simulation objects that are in the block state are not considered in the dispatching process (thus being again eligible for dispatching only after exiting the block state). We have run experiments on a 32-core HP ProLiant server equipped with 64GB of RAM and running Debian 6 on top of the 2.6.32-5-amd64 Linux kernel

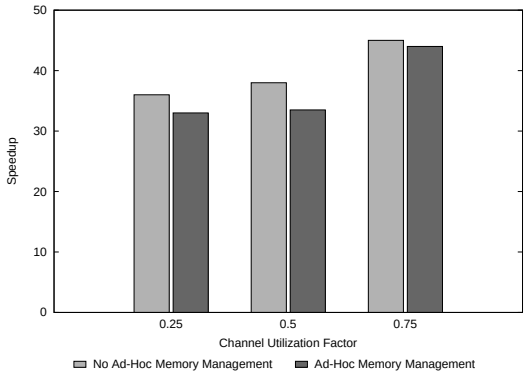

Figure 7: Relative speed-up by the parallel run without and with ad-hoc memory management vs the sequential run.

(augmented with our patch to the `schedule` function). A ROOT-Sim configuration with 32 symmetric worker threads has been used in all the experiments, with GVT and fossil collection taking place each one second.

4.2 Experimental Data

This section is divided in two parts. Initially we provide data for an evaluation of the overhead by the core memory management support underlying ECS. To this aim we use a simulation model of a Personal Communication System (PCS) natively entailing disjointness of memory accesses by the different simulation objects. After, we present data related to the assessment of the whole ECS architecture, by also comparing the run-time behavior of models coded in such a way to be run on top of ECS (hence coded in sequential-style with no disjointness of memory accesses across different objects) with respect to the counterpart exclusively based on traditional PDES programming (only relying on event-dependencies via message passing). In this part of the study we rely on a model of NoSQL data-grid systems developed within the framework presented in [5].

4.2.1 Overhead Assessment

We evaluated the overhead by the presented memory management architecture by relying on a PCS model with 1024 wireless cells covering a square region, each one managing up to 1000 wireless channels. It models interference across different channels within a same cell, and power management upon call setup/handoff in a high fidelity fashion. This same model has been already used to assess the multi-threaded version of ROOT-Sim where we have integrated ECS, and its detailed description can be found in [26]. Two specific aspects are relevant for this study: 1) each simulation object models an individual cell, and the interactions between objects exclusively take place via handoff events of mobile devices across different cells (hence memory accesses by the different simulation objects are intrinsically disjoint); 2) the average granularity (CPU requirement) of the events is directly proportional to the wireless channel utilization factor, since the more channels are busy, the more complex is the calculation of interference and Signal-to-Interference Ratio (SIR) while simulating power regulation.

We have run this model with three different settings for the channel utilization factor, namely 25%, 50% and 75% (that gave rise to average granularity of the simulation events ranging from the order of 30 to 100 microseconds). Also, we considered three different execution modes: a classical sequential execution (relying on a calendar queue scheduler), a parallel execution where no ad-hoc memory management facility is activated, and a parallel execution where we rely on the innovative memory management architecture. Note that the latter execution mode entails switching between object-mode and platform-mode (with refill of the `CR3` register and implicit squash of the TLB) when changing the actual mode. Hence, such a mode allows us to assess the overhead for mode-switch operated by the support for ECS. In Figure 7 we show the variation of the speedup (vs the sequential run) we observed for simulating on the order of 1 million (committed) events in the different parallel execution modes (each sample is the average over 10 runs based on different random-generation seeds). By the data we see how the maximal loss in performance by the ad-hoc memory management architecture entailing switch between platform and object modes is on the order of 9% and is observed for the case of finer grain simulation events (namely for the case of 25% utilization factor). Such a performance penalty almost disappears for coarser grain configurations.

Successively we modified the PCS model in order to generate fictitious rendez-vous events periodically. When one fictitious rendez-vous event occurs, the executing object simply performs a dummy read operation into the state of an adjacent cell. However, we do not really enable ECS synchronization (in fact, no matter whether the dummy read access is not processed in timestamp order on the hit object), rather we only trap the access and open the stock associated with the object hit by the read operation. This way we are able to assess the overhead by ECS support when also including the management of memory faults and the activation of the ECS handler. For this experiment, we considered the PCS configuration with wireless-channel utilization factor set to 50%, and we varied the frequency of occurrence of the fictitious rendez-vous events between 1% and 10% of the total number of events processed. In Figure 8 we show the relative speedup achieved by the configuration with ad-hoc memory management and fault handling upon the occurrence of fictitious rendez-vous events vs the configuration with no ad-hoc memory management. By the data, we see how the ah-hoc architecture induces a speed-down that increases vs the frequency of fictitious rendez-vous events. Note that the speed-down is not only caused by the overhead for handling the memory faults. It is also due to the switch between platform and object modes, which is mandatory in order to create the per-thread memory view needed to trap the access to the state of other simulation objects. However, the speed-down is quite limited for relatively infrequent fictitious rendez-vous events, and becomes non-negligible only when moving towards scenarios with relatively frequent rendez-vous occurrences (say 10%).

On the other hand, the whole ad-hoc memory management architecture has been thought and realized to provide, transparently to the application code, a unique innovative support for handling cross-state dependencies in presence of concurrent objects. Hence the loss in performance in contexts where the model to be executed exhibits intrinsically disjoint accesses across different object-states (such as for the configurations in Figure 7) is the unavoidable price to be paid for the achievement of a run-time environment offering the above mentioned level of transparency.

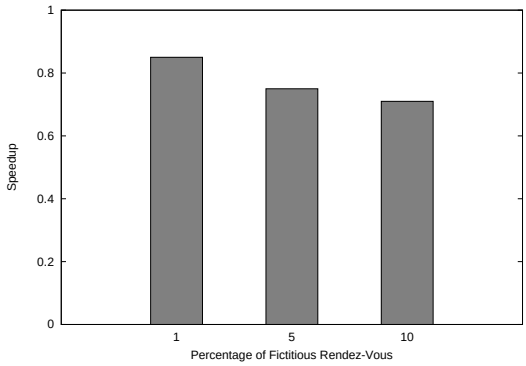

Figure 8: Relative speed-up by the ad-hoc memory management architecture vs the classical parallel run.

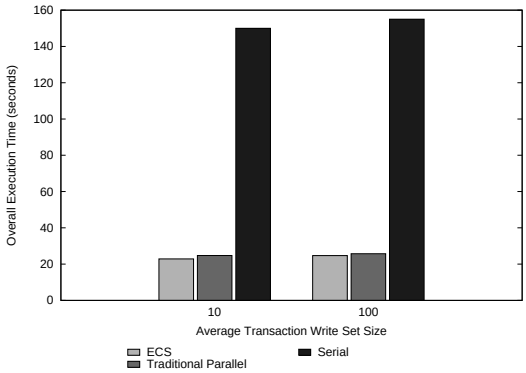

Figure 9: Execution times for the data-grid models.

4.2.2 *Effectiveness Assessment*

In this section we rely on NoSQL data-grid simulation models provided by [5], based on distributed/replicated cache serves, each keeping a subset of the whole set of keys in the entire data-set. Particularly, we consider a model where atomicity of the distributed transactions is ensured by running the 2-Phase-Commit (2PC) protocol across the nodes keeping keys that belong to the write set of the committing transaction. In these models, the simulated coordinator needs to schedule the arrival of a *prepare request* event to the involved sites, which needs to carry information about the write set. These sets may entail hundreds of data-item keys, and are populated at the coordinator while simulating the execution of the transaction. These sets are therefore instantiated by the transaction-coordinator simulation object within its local state. For this model we consider two different implementations, one not relying on ECS, which transmits the write set as the payload of the *prepare request* event[3], and another one based on ECS, where the write sets are directly accessed via pointers by the involved simulated nodes (hence the *prepare request* event only needs to carry the pointer indicating where to find the information related to the simulated 2PC phase). This model also entails a special simulation object which is a global statistics collector. For the case of non-reliance on ECS, the updates of the state of this object take place by explicitly scheduling *update statistics* events towards it. On the other hand, for the case of ECS synchronization, we have that each simulated node can directly access the state of the statistics collector simulation object in order to perform updates.

We simulated a NoSQL data-grid system with 64 nodes (with degree of replication 2 of each $\langle key, value \rangle$ pair in the data-grid), with closed-system configuration in terms of number of clients (and hence number of transactions) running within the system. Particularly, we set the number of active concurrent clients continuously issuing transactions to 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, we varied the amount of keys touched in write mode by transactions between 10 and 100, which gives rise to different dynamics/cost in terms of pack/unpack operations, message buffering and transmission for the case of non-ECS based synchronization, thus allowing us to study configurations with different performance tradeoffs.

In Figure 9 we report the execution time for simulating a predetermined simulation time interval for the operativity of the NoSQL data-grid system. By the data we see how both ECS and the traditional message passing based parallel approach provide performance improvements over the sequential run. More important, the performance delivered by ECS is even slightly better that the one by the traditional parallel approach with disjoint accesses to the local state of the simulation objects. Overall, transparency of speculative parallelization with cross-state dependencies is achieved by also delivering performance comparable to the traditional parallel approach, which however does not mask message passing to the programmer, at least in relation to packing/unpacking of data to/from event payloads. Also, the traditional approach does not support direct memory writes into, e.g., the statistic collector object, thus requiring more complex coding schemes, aimed at realizing the updates via simulation events. On the contrary, with ECS we support such a direct memory update operation, hence offering to the programmer the possibility to code his model more simply, according to sequential programming style.

[3]For this configuration the programmer is in charge of explicitly coding the pack/unpack of the write set.

5. CONCLUSIONS

In this article we have presented ECS (Event and Cross-State), a new protocol for synchronizing the execution of concurrent simulation objects forming a DES model. The protocol allows breaking the classical limit of DES models, to be run concurrently on top of PDES platforms, where any object can access memory, and hence can touch the current state of the simulation model, limited to its local state. Rather, with ECS it is allowed to touch (in either read or write mode) any valid memory location. This capability has been achieved thanks to the design and implementation of an innovative memory management architecture, suited for Linux systems, which creates per-thread views of memory protection (within a same process) and tracks memory accesses in an efficient manner. The whole proposal supports cross-state access, joint to concurrency and speculative processing, in an application transparent manner. Hence, the programmer is allowed to rely on a sequential-style coding approach, where any memory location is implicitly accessible while processing any simulation event.

6. REFERENCES

[1] F. Antonacci, A. Pellegrini, and F. Quaglia. Consistent and efficient output-streams management in optimistic simulation platforms. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 315–326. ACM, 2013.

[2] D. Bruce. The treatment of state in optimistic systems. *SIGSIM Simul. Dig.*, 25(1):40–49, July 1995.

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

[4] 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 IEEE Workshop on Principles of Advanced and Distributed Simulation*, pages 1–9. IEEE Computer Society, 2011.

[5] P. Di Sanzo, F. Antonacci, B. Ciciani, R. Palmieri, A. Pellegrini, S. Peluso, F. Quaglia, D. Rughetti, and R. Vitali. A framework for high performance simulation of transactional data grid platforms. In *Proceedings of the 6th ICST Conference of Simulation Tools and Techniques*, SIMUTools, pages 63–72. ICST, Mar. 2013.

[6] 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*, pages 82–89, jun 1997.

[7] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

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

[9] K. Ghosh and R. M. Fujimoto. Parallel discrete event simulation using space-time memory. In *Proceedings of the International Conference on Parallel Processing*, pages 201–208. CRC Press, 1991.

[10] HPDCS Research Group. ROOT-Sim: The ROme OpTimistic Simulator - v 1.0. `http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/`, Oct. 2012.

[11] D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev. Optimization of parallel discrete event simulator for multi-core systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 520–531. IEEE Computer Society, 2012.

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

[13] V. Jha and R. Bagrodia. Simultaneous events and lookahead in simulation protocols. *ACM Transactions on Modeling and Computer Simulation*, 10(3):241–267, July 2000.

[14] J. I. Leivent and R. J. Watro. Mathematical foundations of Time Warp systems. *ACM Transactions on Programming Languages and Systems*, 15(5):771–794, 1993.

[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*, 82(8):511–521, Aug. 2006.

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

[17] H. Mehl and S. Hammes. How to integrate shared variables in distributed simulation. *SIGSIM Simulation Digest*, 25(2):14–41, Sept. 1995.

[18] D. M. Nicol and X. Liu. The dark side of risk (what your mother never told you about time warp). In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pages 188–195. IEEE Computer Society, 1997.

[19] A. Pellegrini, R. Vitali, S. Peluso, and F. Quaglia. Transparent and efficient shared-state management for optimistic simulations on multi-core machines. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 134–141. IEEE Computer Society, 2012.

[20] A. Pellegrini, R. Vitali, and F. Quaglia. Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation*, pages 45–53. IEEE Computer Society, 2009.

[21] B. R. Preiss, W. M. Loucks, and D. MacIntyre. Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253, July 1994.

[22] F. Quaglia. A cost model for selecting checkpoint positions in Time Warp parallel simulation. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):346–362, Feb. 2001.

[23] R. Rönngren and R. Ayani. Adaptive checkpointing in Time Warp. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pages 110–117. Society for Computer Simulation, July 1994.

[24] A. Santoro and F. Quaglia. Software supports for event preemptive rollback in optimistic parallel simulation on myrinet clusters. *Journal of Interconnection Networks*, 6(4):435–457, 2005.

[25] R. Toccaceli and F. Quaglia. DyMeLoR: Dynamic Memory Logger and Restorer library for optimistic simulation objects with generic memory layout. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation*, pages 163–172. IEEE Computer Society, 2008.

[26] R. Vitali, A. Pellegrini, and F. Quaglia. A load sharing architecture for optimistic simulations on multi-core machines. In *Proceedings of the 19th International Conference on High Performance Computing*, pages 1–10. IEEE Computer Society, Dec. 2012.

[27] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation*, pages 211–220. IEEE Computer Society, Aug. 2012.

[28] D. West and K. Panesar. Automatic incremental state saving. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pages 78–85. IEEE Computer Society, May 1996.