Approximated Rollbacks

Matteo Principe
matteo.principe@students.uniroma2.eu
University of Rome “Tor Vergata”
Rome, Italy

Andrea Piccione
piccione@diag.uniroma1.it
Sapienza, University of Rome
Rome, Italy

Alessandro Pellegrini
pellegrini@diag.uniroma1.it
Sapienza, University of Rome
Rome, Italy

Francesco Quaglia
francesco.quaglia@uniroma2.it
University of Rome “Tor Vergata”
Rome, Italy

ABSTRACT

A rollback operation in a speculative parallel discrete event simulator has traditionally targeted the perfect reconstruction of the state to be restored after a timestamp-order violation. This imposes that the rollback support entails specific capabilities and consequently pays given costs. In this article we propose *approximated rollbacks*, which allow a simulation object to perfectly realign its virtual time to the timestamp of the state to be restored, but lead the reconstructed state to be an approximation of what it should really be. The advantage is an important reduction of the cost for managing the state restore task in a rollback phase, as well as for managing the activities (i.e. state saving) that actually enable rollbacks to be executed. Our proposal is suited for stochastic simulations, and explores a tradeoff between the statistical representativeness of the outcome of the simulation run and the execution performance. We provide mechanisms that enable the application programmer to control this tradeoff, as well as simulation-platform level mechanisms that constitute the basis for managing approximate rollbacks in general simulation scenarios. A study on the aforementioned tradeoff is also presented.

CCS CONCEPTS

• **Computing methodologies → Discrete-event simulation**; **Simulation environments**; *Massively parallel and high-performance simulations*; *Distributed simulation.*

KEYWORDS

State Saving, Rollback Operation, Speculative Simulation, Time Warp

ACM Reference Format:
Matteo Principe, Andrea Piccione, Alessandro Pellegrini, and Francesco Quaglia. 2020. Approximated Rollbacks. In *Proceedings of the SIGSIM Principles of Advanced Discrete Simulation (SIGSIM-PADS '20), June 15–17, 2020, Miami, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3384441.3395984

1 INTRODUCTION

Parallel Discrete Event Simulation (PDES) [4] enables the exploitation of the scaled-up computing power offered by parallel and distributed platforms to reduce the execution time of simulation runs and to make large/huge simulation models tractable. It is based on partitioning the entire model to be simulated in simulation objects—historically referred to as Logical Processes (LPs)—which are CPU-dispatched for processing their events concurrently. A challenge in this research area is to keep the concurrent evolution of these objects consistent, and the classical reference criterion states that the overall simulation run is consistent if each object processes its incoming events—possibly produced by other objects—in non-decreasing timestamp order.

A scalable approach to ensure timestamp-ordered execution of events at each individual simulation object is *optimistic synchronization* [7]. This solution is based on speculative processing techniques, where objects are allowed to process their events without any assurance that timestamp order violations will not occur. Hence no cost is paid to a-priori determine the safety of the events to be processed. Violations can occur, as an example due to the arrival at object O of some newly-scheduled event from a concurrent object O', with timestamp in the past of O's already processed events. If a violation occurs, then the simulation object affected by the *causality error* (in our case O) rolls back in logical time. This leads to the restoration of the last state value not affected by the violation.

The common way to support the rollback operation of a simulation object is to perform an exact reconstruction of its last correct state, and many literature studies have investigated solutions to perform this operation at reduced cost (e.g. [2, 14]). Most of these solutions take into account the tradeoff between the overhead caused by the activities that enable the capability to rollback—such as saving state information along the forward execution of the simulation object—and the actual overhead caused by the reconstruction phase, which squashes the current inconsistent state in favor of an older still-consistent state image.

In this article we investigate an orthogonal approach to the reduction of the cost for rolling back the simulation object to a past state, which we name *approximated rollback.* Our solution is based on the idea that the perfect reconstruction of the last correct past state when a causality error takes place might be “redundant” to achieve a meaningful execution of the simulation. In more details, we may get acceptably accurate simulation results even in scenarios where the resume of the execution of the causality-error affected simulation object takes place from a state that is a reasonable approximation of the one that should have been restored. Clearly, this solution can fit stochastic simulations, where we typically estimate properties of the simulated system through statistical techniques.

The central concept that stands behind our solution is that the state of a simulation object can be partitioned into *core* and *non-core* portions. The core portion allows reconstructing the whole state, although in an approximated way. This allows focusing on the core portion only, when both state saving and state reload from the log need to be carried out. On the other hand, the possibility to devise a portion of the simulation object state as core is application-dependent, hence the approximated rollback technique requires a bit more intervention by the application programmer in terms of interaction with the state management logic offered by the optimistic runtime environment. To reduce the need for intervention by the programmer, we provide a state-management architecture[1] that allows to transparently save the core-state portion, and supports an application-level callback which, starting from a saved core-state portion, will *approximately* reconstruct the whole state when a rollback occurs. Our architecture also offers the possibility to dynamically change the identification of the core-state portion, which allows to further optimize the execution of the approximated rollback technique when the simulation object state dynamically changes in shape and semantics.

In our architectural proposal we still enable the coexistence of approximated rollback phases and traditional non-approximated ones, based on the perfect reconstruction of past states. Also, we leave to the simulation model the possibility to switch between the two at runtime. The final effect is to enable the simulation model to choose what phases of the simulation run can tolerate approximations and what phases cannot. This choice can depend on the runtime observation of the actual state of a simulation object and on its evolution trajectory, so that the approximated mode can be enabled or disabled depending on whether specific predicates hold or not. Ultimately, the choice on whether to enable or disable this support can be simply based on the passage of logical time, and on the willingness to allow no more than a given percentage of passed-through logical time to be managed with approximated rollbacks.

The architectural support of the approximated rollback technique has been implemented in the open-source ROOT-Sim optimistic PDES platform [10] and is therefore available for download[2]. Experimental data for an assessment of our proposal are reported in this article, in particular for the case of a synthetic benchmark based on PHold [5], and a real-world agent-based epidemiological Susceptible-Infected-Recovered (SIR) model [8], which allows to study the spreading of viral diseases.

The remainder of this paper is structured as follows. In Section 2 we discuss related work. The approximted rollback technique and its support are presented in Section 3. Experimental data are provided in Section 4.

[1]Overall, it can be regarded as a rollback-capable memory management architecture.
[2]The official ROOT-Sim repository is at https://github.com/HPDCS/ROOT-Sim. The artifact submitted for reproducibility of this work is available at https://doi.org/10.5281/zenodo.3765238.

2 RELATED WORK

The objective of reducing the cost for managing state-rollback operations in speculative PDES has been long studied in the literature. A few solutions propose to exploit infrequent state snapshots—to reduce the CPU-cost and memory footprint for logging state information—and to restore a missing snapshot by reloading a previous one and reprocessing intermediate events. In these solutions, a key point is to optimize the joint costs of taking the snapshots and rolling back to the target state image via the intermediate reprocessing phase. Several methods to determine the best-suited snapshots to take, or how frequently shapshots should be taken, have been proposed (see, e.g., [14, 15]).

Other approaches are based on taking incremental checkpoints, which only log the portions of the simulation-object state that have been modified since the last checkpoint operation (see, e.g., [18, 21]). This reduces the CPU-cycles and memory footprint to create the checkpoints along the forward execution phase. However, the rollback phase requires to backward apply all the incrementally logged checkpoints to enable the resume of the simulation object execution from the correct past state. Sometimes, the length of the sequence of backward steps is unpredictably long, leading to negative performance effects. To cope with this problem, some solutions [12, 19] have proposed mixed schemes where incremental and non-incremental checkpoints are taken in interleaved manner along the forward execution phase of a simulation object.

The extremism of the reduction of the memory footprint associated with state saving has lead a few authors to rely on reverse computing approaches [1]. These are based on backward reprocessing (in reverse order and manner) all the causally-inconsistent events upon the rollback task. Except for irreversible state changes, with this solution no logs of state information is required. However, for (very) long rollbacks the cost of backward reprocessing the events might be a dominating factor adverse to performance. To cope with this issue, the work in [2] has studied how to mix reverse computing with classical infrequent checkpoints, so as to limit the maximum number of backward-processed events in a rollback phase.

All the aforementioned approaches target the scenario where the reconstruction of the past state to be restored is "perfect", meaning that the restored state is an exact copy of the corresponding state observed along the forward execution phase, before the occurrence of the causality error. In this paper, we take the different approach of reconstructing the state to be restored in an approximated manner, hence enabling a new dimension in terms of the tradeoff between the CPU-cost and the memory footprint to save state information, and the latency to restore the last correct state upon rolling back. As for transparency, our solution requires some intervention by the programmer, who needs to include in the application code specific calls to an API that enables the approximated rollback scheme, and needs to implement a callback function to exploit the platform-level state-saving tasks in order to approximately rebuild the state to be restored. However, following a few guidelines we shall introduce, the above requirements affect the complexity of coding the application logic of a PDES application in a relatively limited manner. Also, the additional logic the programmer needs to put in the coded model is domain-specific and fully bound to the features of the coded model. In other words, the programmer does not need to be aware of what a rollback is. He only needs to be aware of how the state of a simulation object can be reshaped starting from a minimal set of state information transparently managed (i.e., logged and restored) by the underlying PDES platform. In any case, a few of the above cited solutions do not provide the support for full transparency to the application programmer (e.g. [14, 15, 18, 19]).

Our proposal is also related to solutions based on the notion of uncertainty in the occurrence of events [3, 16]. In these proposals, the PDES platform can take decisions aimed at reducing the execution time of the simulation by exploiting a kind of "under-specification" of either time and space for the occurrence of events. The final effect can be a bias in the evolution of the simulation model trajectory that can ultimately lead to an approximation of the collected statistics—compared to a scenario where no under-specification is used and exploited by the runtime system. However, a core difference in our solution is that the effects of the approximation in the model-execution trajectory, caused by approximated rollbacks, can be somehow controlled by the application programmer, since the execution phases where the approximated rollbacks can occur are defined by the application logic. Also, with the approximated rollback scheme, the programmer has the ability to determine what portions of the simulation-object state define the core, and what portions are instead less valuable, and can be rebuilt in an approximated manner. This is an additional means to provide the programmer with a way to control the effects of approximated rollbacks on the output statistics by the simulation.

Other proposals have studied the tradeoff between relaxing strict causality of the events and its effects on performance in (speculative) PDES [6, 17]. Essentially, these solutions skip running some state rollbacks if the effects of processing events out of strict timestamp order is considered acceptable—in terms of the final statistics computed by running the simulation. Our solution can be seen as fully orthogonal to these approaches, since we still comply with strict timestamp ordering of the events at each individual simulation object, but we allow a given event to observe an approximation of the state that would have been observed in a rollback-free run of the same simulation model.

3 APPROXIMATED ROLLBACK TECHNIQUE

In this section we initially discuss the core aspects associated with the approximated rollback technique, and its implications on the application coding/configuration process. Then we present the architectural development we carried out to integrate the support for approximated rollbacks in the open source ROOT-Sim platform.

3.1 Outline

The approximated rollback technique deals with the ability of a speculative PDES application to resume its execution, after the occurrence of a rollback that brings a simulation object back to logical time T, from a state S' which does not fully match with the original state S that was observed by the simulation object (at the same time T) along the forward-mode execution.

Denoting with $VT(x)$ the virtual time of a state x, and with $I(x)$ the actual information kept by x—namely the collection of state-variables forming the state—and denoting with $S' \longleftarrow S$ the relation between two states S' and S such that S' represents the restoration of S after a rollback, then the approximated rollback technique can be characterized through the following expression:

$$S' \longleftarrow S \Rightarrow [\ VT(S') = VT(S)\ ] \ \wedge [\ I(S') \subseteq I(S)] \tag{1}$$

By Equation 1, the logical time of the two states S' and S involved in the $S' \longleftarrow S$ relation is identical, while S entails a superset of the information kept by S'. In other words, the approximated rollback leads to restore a state with some missing information, but there is no added information (i.e. no added state-variable) in S' that was not originally present in S. This is a crucial aspect from the point of view of model coding, since the application programmer knows that after an approximated rollback takes place he will never need to deal with a state-variable that was not already observed along the forward execution mode.

As for model execution in a speculative PDES environment, the subset relation between $I(S')$ and $I(S)$ implies that, in order to create a snapshot useful to restore S', when we pass through S we can log (much) less data with respect to the size of S. We know that logging less data than the state size to create a snapshot is a classical achievement of many incremental state saving techniques. However, in the approximated rollback technique, in order to reduce the log size (and the CPU-time to take it) there is no need to relate data that are saved in the log—and that are used to restore the object to state S'—with the write operations occurring on the object state along the forward execution mode. We simply need to discard some information belonging to S to create the log useful to restore S'. This also means that we do not need to put in place any tracing mechanism (based on, e.g., software instrumentation) to track state updates to reduce the log size. Also, we do not need to backward re-traverse the incremental log chain to rebuild S. We simply restore the saved *core information* of S—namely $I(S')$ rather than $I(S)$—which enables restoring S' in place of S.

Another implication of the relation $I(S') \subseteq I(S)$ is that S' is not guaranteed to fully record the *history* of the events that have been processed at the simulation object up to the restoration time T. In fact, some information that these events installed in the originally passed-through state S is no longer there after rolling back to state S'. Accepting this risk implicitly leads to discarding the idea that reducing the amount of per-event logged state information in the speculative PDES run should be achieved via sparse state saving techniques combined with coasting forward. In fact, a coasting forward phase is targeted at the regeneration of a state image that exactly records the whole history of events that are processed and not affected by a causality violation, including the events that are intermediate between the logical time of the last checkpoint before the causality error, and the timestamp T of the last correct event. Avoiding this type of "perfect state reconstruction" makes the state restoration time upon a rollback independent of the granularity of the events.

Overall, dealing with large state sizes and coarse-grain events (possibly updating large portions of the state) in a speculative PDES application is no longer a concern when employing approximated rollbacks, especially when $I(S')$ is a very reduced subset of $I(S)$. On the other hand, large state sizes and coarse-grain (write-intensive)

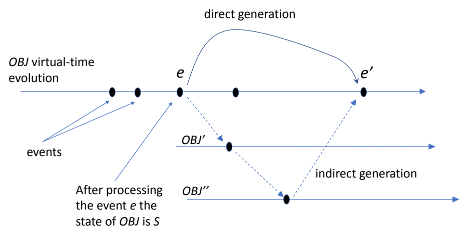

direct generation
OBJ virtual-time evolution
e
e'
events
OBJ'
OBJ''
indirect generation
After processing the event e the state of OBJ is S

Figure 1: Direct and indirect generation of e' by e.

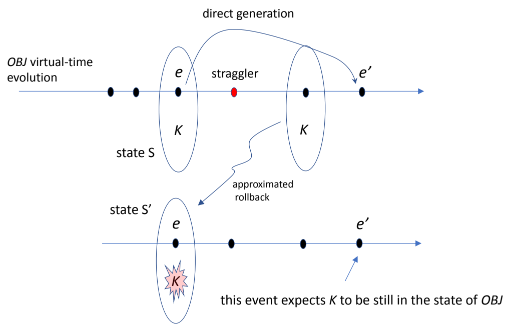

direct generation
OBJ virtual-time evolution
e
straggler
e'
K
K
state S
approximated rollback
state S'
e
e'
K
this event expects K to be still in the state of OBJ

Figure 2: The runtime exception (key not found) caused by the processing of e'.

events represent a challenging scenario for all the other techniques, including reverse-computing based ones[3].

Clearly, in our approach, we also need to consider the feasibility of resuming from S' (rather than S) after a rollback in terms of correctness of the simulation model execution. With *correctness* we do not mean the statistical outcome of the stochastic simulation run, which can be clearly affected by an approximation in the state restore phase. Rather, we intend that the processing of the events occurring after an approximated rollback does not give rise to exceptions that lead to the abort or a crash of the simulation run. If it were the case, then we would give no guarantee that a speculative PDES simulation based on approximated rollbacks can eventually reach its termination.

The problem to be considered is still related to the fact that S' will result in a *partial* record of the history of events processed at the simulation object, say OBJ, up to the restoration time T. But we know that the history of events occurring up to time T at the object OBJ can affect the future of the same object, either directly or indirectly. Looking at the examples in Figure 1, a direct effect takes place when an event e with timestamp $T(e) < T$, executed by OBJ, schedules a new event e' with timestamp $T(e')$ for the same object, with $T < T(e')$. On the other hand, an indirect effect appears when the event e leads to schedule e' passing through other simulation objects—so e leads to the scheduling of one or more intermediate events across a chain of simulation objects, which ends with the scheduling of the event e' at object OBJ.

The key point here is that the event e has access to state information related to the history of the events processed at object OBJ up to $T(e)$, and the newly-scheduled event e' can have a payload for which we may expect some match with the information found by e in the object state when it was processed. A classical example is that e installs in the object state a key K at time $T(e)$, which will need to be removed by another event e' in the future at time $T(e')$, with e' scheduled by e. Suppose this information—namely the key K—is still included in the state S when OBJ transits from $T(e)$ to T, and then disappears because an approximated rollback at time T (which does not undo the execution of e) restores state S' rather than S. In this case, when moving to $T(e')$ (with $T(e') > T$) after the post-rollback resume, the event e' will generate an exception (the key is not found in the object state). An example of this situation is depicted in Figure 2.

The above discussion allows us to introduce the concept of combination between approximated and non-approximated rollbacks, to be used to avoid such runtime anomalies. When an event e is processed such that a new event e' is scheduled (or it is simply expected to be scheduled) in the future, whose processing will not tolerate an approximation in the state evolution trajectory, then the simulation model can simply switch to traditional rollbacks. As for the above example, at time $T(e)$ the model can switch to non-approximated rollbacks, and then can switch back to approximated rollbacks a time $T(e')$. Outside this logical-time window, approximated rollbacks can work with no risk of runtime exceptions. Otherwise, we can adopt the different approach where the key K is kept in the core of the state of the object—hence also in the core of the state S—so we know that we still find this key into $I(S')$ when S' is restored in an approximated rollback.

A last point needs to be discussed to conclude the outline of the approximated rollback technique. It is related to the following question: «*once we restore S' (instead of S), are we able to perform a guess on the missing piece of information between $I(S)$ and $I(S')$?*». Considering stochastic simulation as the target, the answer is yes. In particular, we can complement the restore of S' with the invocation of an application-level callback that simply takes S' and changes its form in order to (possibly) improve the real distance that exists between $I(S)$ and $I(S')$. Overall, denoting with CF this callback function, we have that the final state that is restored at time T in an approximated rollback is $CF(S')$. Clearly, the inclusion of the callback function CF to support the approximated reconstruction of the state from which to resume the processing of the simulation object after a rollback adds some runtime overhead to the state reconstruction process. However, we still avoid the dependence between the cost of the approximate state reconstruction and the amount of rolled back events.

At this point we can add another indication on how to build S', which is the actual core of S, in order to make the callback function CF effective: S' (and hence S) should include information on what pieces of the state are missing between S and S'. However, their values are a "do not care", and will be simply regenerated (in an approximated manner) by the callback function CF. Clearly, it is

[3]In fact, coarse-grain events with many updates on the state will likely lead to coarse-grain backward computing steps, independently of whether the backward computing phase is based on reverse event handlers [1] or reverse reconstruction of memory-location values [2].

left in the hand of the application programmer to determine what information is suited for the approximate reconstruction via CF, and what is instead more relevant to the stochastic process we are simulating, and should be therefore maintained in the core of S—so that we are sure that we will find it again in S' after an approximated rollback. In the next section we provide an architectural support for approximated rollbacks and a general indication on what the approximated rollback support can provide to make the job of the programmers easier. On the other hand, the above described solution is an archetypal that can be instantiated in differentiated modes, also depending on the specific software technology used to develop models and/or to implement the speculative PDES runtime environment.

3.2 Architectural Support

We have instantiated the approximated rollback support integrating it in the ROOT-Sim speculative PDES platform. This platform allows running discrete event models developed using the C programming language. Hence, the exploitation of the capabilities of ROOT-Sim enables the programmer to build simulation objects with a strict control on the memory layout of their states. In more details, a simulation object in ROOT-Sim is represented by a set of memory buffers which are dynamically allocated (and possibly released) along the lifetime of the object, using the conventional functions `malloc` and `free` offered by the standard library of the C programming language. Also, when an object is scheduled for event processing, the event-handler callback that is invoked by the runtime environment receives as input parameter a pointer that allows accessing a chunk of memory representing the "top level" state information. Starting from this chunk, any other chunk currently allocated in the object state can be still reached via pointers.

The set of memory chunks forming the object state represents the so called "memory map" of the object. To enable state restoration, various mechanisms are offered, ranging from infrequent to incremental checkpointing of the memory map. In particular, the memory map of each object is handled via the DyMeLoR [11] layer, which keeps compact metadata for identifying allocated chunks, dirty chunks and chunks whose speculative release is not yet committed (hence it can still be undone). To keep these metadata consistent, DyMeLoR transparently intercepts the calls to `malloc` and `free` as natively issued by the application-level code. Also, it can intercept the memory write operations issued by the application level code through binary instrumentation schemes still operating transparently to the application-code developer. A checkpoint operation involves copying the currently-allocated chunks into a log buffer, while the restore operation puts the logged chunks back in place in their original memory locations.

In this memory management model, the information $I(S)$ associated with a state S observed by the simulation object along event processing is formed by two parts:

1) the collection of the memory addresses associated with the currently active chunks, and their size—we can globally define them as the chunk areas;
2) the actual content of the currently active chunks.

Considering a state S', which is an approximated restore of S, then $I(S')$ might miss a subset of the chunk contents—but it might still keep the chunk areas—and/or might miss a subset of the chunk areas—hence also missing their content.

In the currently developed support for approximated rollbacks, we opted for the second choice. Therefore, S' is an approximation of S because the actual memory map associated with S' misses memory chunks that were instead present in the memory map associated with S—at the time when S was passed through by the object along the forward-mode execution of the simulation. In other words, we do not allow to restore a chunk of memory in the memory map of S' without restoring its content.

This approach allows us to define the concept of strongly connected core memory in the object state as follows:

Definition 3.1 (strong core-memory connectivity). All the pointers between any couple of chunks of memory $\langle A, B\rangle$ that belong to the core of the state S, namely the portion of S which is restored in an approximated rollback through the restoration of S', are still meaningful—they are not dangling pointers.

It is important to note that this property deals with the mutual relation (through pointers) between couples of memory chunks that are in S', while it does not apply to an individual memory chunk. In other words, with strong core-memory connectivity, it is still possible that a chunk belonging to S' has a dangling pointer—a pointer that points to a memory address not covered by any chunk of S'. Essentially this is a missing chunk in the approximated rollback.

To better illustrate this aspect, let us consider the scenario depicted in Figure 3. In the top part we represent the organization of the state S, when it was passed through in forward-mode execution. As for the content of the different chunks included in the memory map of the object state S, we focus on pointer fields possibly kept within locations included in the chunks. In the bottom part of the same picture we show the organization of S'. Here, all the buffers that are marked red in the layout of S no longer appear in the layout of S'—but those that are in S' have identical content with respect to the corresponding ones that were in S. Therefore, we correctly have that $I(S') \subseteq I(S)$. For the sake of simplicity in the discussion, we marked the different memory chunks with labels. Also, with no loss of generality, in the example scenario we have assumed that the memory chunk A represents the top-level state information pointed to by the pointer passed to the event-handler routine when an event is CPU-dispatched at the simulation object—this pointer is denoted as P_0 in our representation. By the example, we have that strong core-memory connectivity is respected when restoring S' instead of S, although chunk E has a dangling pointer p_k. The same is true for the pointer p_h kept in the memory chunk A.

Now, the relevance of the strong core-memory connectivity property is that we can rely on a simple approach, to be followed at the application level when coding the state management logic of the simulation object, in order to enable restoring a state S' such that, starting from the pointer P_0, we can be able to actually access all the chunks that are in the memory map associated with S'—this means that S' is fully reachable starting from P_0. The approach is that for any couple of chunks $\langle A, B\rangle$ belonging to the state S, such that there is a direct pointer-based relation between the two, namely $A \xrightarrow{p} B$, then B can be part of the core memory of state S if and only if A is in the core memory too. The only exception is

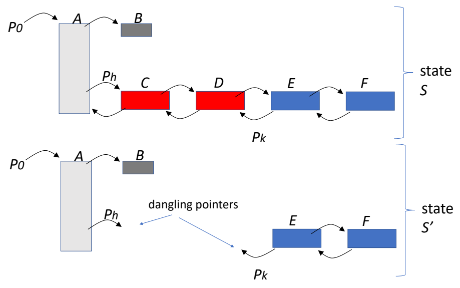


Figure 3: Explaining the strong core-memory connectivity property.

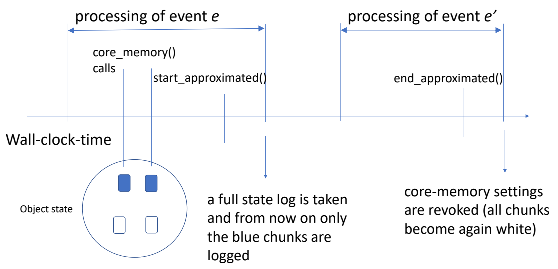

processing of event e
processing of event e′
core_memory() calls
start_approximated()
end_approximated()
Wall-clock-time
Object state
a full state log is taken and from now on only the blue chunks are logged
core-memory settings are revoked (all chunks become again white)

Figure 4: An example timeline.

for the top-level chunk, namely chunk A pointed to by P_0 in our example.

We recall that, in the approximated rollback scheme, we also include the possibility to use the callback function CF, which receives S' in input, in order to perform some transformation of S' possibly leading to a better approximation of the original state to be restored S. Clearly, full reachability of S' is a fundamental aspect to effectively code CF. However, another important point to correctly manage the core information available in S' is to make the callback function CF able to deterministically recognize dangling pointers embedded in chunks belonging to S'. We recall again that, following the above approach to define what chunks belong (and what chunks do not belong) to the core-memory portion of the state S—so as to build a fully-reachable set of chunks—a dangling pointer in a chunk included in the restored state S' will surely point to a chunk that has been missed in the approximated rollback.

The support for the deterministic identification of what pointers are not dangling is the inclusion of a new type of data structure, called `struct discriminable_pointer`, such that the virtual address to be used to point to a chunk in the object state S is coupled with a control bit-mask telling whether it is pointing to a chunk in the core-memory part or not—in the positive case we say that it is a core pointer. In this way, the callback function CF can safely scan all the buffers in S' via pointers, deciding whether to materialize or not missing chunks (those pointed to by non-core pointers) using stochastic values for filling their fields, which are somehow representative (or feasible) for the specific model that is being simulated. In our implementation we provide macros, available to the application programmer, to set a `discriminable_pointer` variable to a pairs of $\langle address, type \rangle$ where the type field codes the control bits corresponding to either a core pointer or a non-core pointer. Other macros are available to simply read the pointer value (namely the address kept by the pointer) and its state (core vs non-core) or for switching its state from core to non-core and vice versa.

At the simulation engine level, the DyMeLoR memory-map manager has been augmented with the new API function `core_memory( void *address, int command)`, which can be used to indicate to the runtime if a chunk with a given address needs, from now on, to be included in the core part of the object state. The `command` parameter can have two values (`INCLUDE` or `REMOVE`) so that the application programmer can decide to dynamically include or exclude a chunk from the core of the object state along model execution, depending on what is going on at the application level. Clearly, all the invocations to `core_memory()` are rollbackable, so they can be revoked transparently by the DyMeLoR memory-map manager. This implies that, at each simulation time T, the chunks that are seen as belonging to the core or not are exactly those that are selected by the application logic up to time T, independently of updates that were possibly performed after time T but before a rollback occurrence to time T. This allows the application programmer to only be faced with the forward execution flow of the model, and with the possibility to define with no ambiguity what parts of the object state are included in the state core memory at any given point in simulation time.

Two additional API functions are used in our architecture, in order to either start an approximated rollback phase or end it—thus resuming to the classical precise rollback scheme. These two API functions need to be used by the programmer to cope with the scenarios depicted in Section 3.1, where at some point in simulation time an event e is processed leading to direct/indirect scheduling of another event e' for the same object, which needs to observe in the simulation object state the effect of the exact history of events up to e. Table 1 shows the complete set of API functions we included in our implementation to support approximated rollbacks.

As a last consideration, we show in Figure 4 an example timeline for the usage of these API functions by the application code. At time $T(e)$ the event e is processed, and during the processing the application code invokes the function `start_approximated()`, after having called `core_memory()` for marking some of the chunks in the state layout as core memory. At this point, a full checkpoint of the whole object state is transparently taken via DyMeLoR and, from now on, only those chunks that have been previously marked as core memory will be subject to checkpoint operations. Then, at time $T(e')$, the application model calls `end_approximated()`, and at this point the difference between core and non-core memory disappears (at the level of the metadata maintained by DyMeLoR) so that the runtime PDES environment starts again adopting a classical checkpoint mechanism where all the chunks belonging to the state layout are logged when a checkpoint is taken. In other words, the `end_approximated()` call revokes the discrimination between core and non-core memory—hence the previously issued calls to `core_memory()` used to set some chunks as core ones are logically invalidated (starting from the current simulation time). Clearly, the

API fuctions
`int core_memory(void *address, int command)`
`void start_approximated(void)`
`void end_approximated(void)`
API macros
`void set_pointer(struct discriminable_pointer, void *address, int type)`
`void set_pointer_type(struct discriminable_pointer, int type)`
`void set_pointer_address(struct discriminable_pointer, void *address, int type)`
`int get_pointer_type(struct discriminable_pointer)`
`void *get_pointer_address(struct discriminable_pointer)`

Table 1: API exposed to support approximated rollbacks.

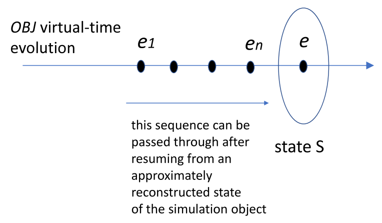
possibility to take checkpoints infrequently is still enabled along a period where the non-approximated rollback mode is active. In fact, a non-checkpointed state can still be perfectly reconstructed by relying on the reload of a previous checkpoint and on coasting forward.

A couple of additional notes need to be discussed. In our architectural support, the callback function *CL* to be defined by the application programmer is called as soon as an approximated rollback occurs, and the state S' is restored. The execution of *CL* takes place before any other event is processed at the simulation object upon resuming after the rollback phase. At this point, the *CL* function needs to find the `discriminable_pointer` variables correctly set. Therefore it is the care of the programmer to correctly mark these pointers and the core chunks via the `core_memory()` API, right before calling `start_approximated()` and for the whole simulation time where the approximated rollback mode is active.

Finally, the usage of `discriminable_pointers` to link the chunks is not mandatory, and the programmer can still rely on classical pointers. In more details, the state S' might entail pointers whose nature (core vs non-core) can be implicitly identified when running the *CF* callback. An example of this scenario is when the top level memory chunk in the object state points to disjoint lists of records, and only a subset of these lists need to be kept in the core of the object state. In this case, the *CF* function can implicitly assume that the lists not belonging to the core need to be simply repopulated (with stochastically-generated values for the content of their records) upon the occurrence of an approximated rollback.

3.3 Further Optimizations

Up to now we discussed the role of approximated rollbacks by considering that the approximation for a rollback that needs to restore the simulation objects at some past simulation time T is essentially caused by the reload of core information only (missing therefore other information), which was originally saved when passing time T along forward execution. However, we can extend this concept by considering that the events processed by the simulation object before reaching time T along forward execution can be part of a simulation portion that can still be correctly handled via approximated rollbacks (see Section 3.1 for the requirements). Let us consider the example in Figure 5, where we have a state S that is passed through by the simulation object after other events $e_1 \dots e_n$ were processed. If the simulation time interval between the timestamp of e_1 and the logical time of state S can be handled by approximated rollbacks, then it means that we can admit the events $e_1 \dots e_n$ to be processed with no runtime anomaly after an approximated rollback that leads



OBJ virtual-time evolution
e1
en
e
this sequence can be passed through after resuming from an approximately reconstructed state of the simulation object
state S

Figure 5: An interval of events manageable in an approximated coasting forward.

to reprocess this same event sequence. If this is true for the processing of these events along forward execution after a rollback, it is still true if we process them as a simple artifact of a state update we would like to perform during a coasting forward phase. This means that for the whole simulation time interval between the timestamp of e_1 and the restore time T at which we would like to reconstruct state S in an approximated rollback, we can use coasting forward starting from an approximately reconstructed state—hence an *approximated coasting forward.* Overall, approximated rollback opens to the possibility to include an additional dimension of freedom for the reduction of the cost for rollback management in speculative executions, based on infrequent core-memory logs.

4 EXPERIMENTAL DATA

In this section we present an experimental study for an assessment of our approximated rollback technique. We initially describe the computing platforms used for the experiments. Then we provide details of the simulation applications that have been used as the test-beds, alongside with the associated results.

4.1 Computing Platforms

We have run experiments on two platforms. The first platform is a bare-metal one, composed of a server equipped with 4 AMD Opteron™ 6168 multicore processors, each one with 12 cores, resulting in a total of 48 cores, and 128 GB of RAM. The second one is a virtualized infrastructure, composed of 3 virtual nodes from Amazon Web Services (AWS). We have relied on three homogeneous compute nodes, namely three m5.8xlarge instances, equipped with 128 GB of RAM and, and 32 virtual CPUs. Virtual instances belonging to the m5 family are run on top of Intel Xeon® Platinum 8175 CPUs, with clock rate up to 3.1 GHz. All instances have been deployed in the same AWS region, and in the same availability zone, thus reducing the interconnection latency.

4.2 Test-bed Applications

Our experiments are based on a synthetic benchmark and a real-world application. The synthetic benchmark is a variation of the traditional PHold benchmark [5], which has been developed according to the specification in [20]. In this version of PHold, beyond the traditional busy loop used to mimic the delay of a simulation event, every simulation object allocates upon the simulation startup a scattered state composed of multiple linked data structures. These

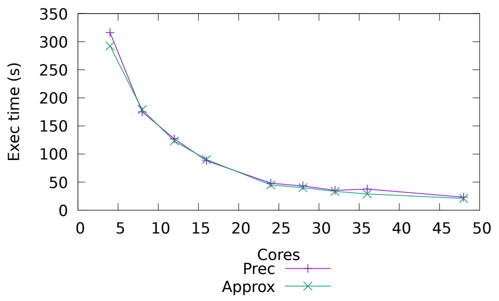


Figure 6: TB epidemic model - total execution time (χ = 10).

data structures are synthetic, in the sense that they keep a member telling their size and a buffer which is filled with random data. While executing an event, the simulation model scans through this data structure and updates the content of a subset of the simulation state, by writing additional random data. With some probability, one of the linked data structures is unchained from the simulation state of an object, and sent towards some random receiver as the payload of a message. Upon the receipt of such message, the destination simulation object installs a copy of the data structure in its simulation state, and continues to maintain its content according to the above-described logic.

This organization of PHold allows to capture additional characteristics of simulation models, related to their memory usage. In particular, a simulation object can have a simulation state which is different from that of other simulation objects (thus mimicking a scenario with different simulation objects representing different real-world entities). Moreover, the data exchange mimics the traditional behavior of PDES applications, according to which simulation objects exchange information among each other. Finally, the overall size of the simulation model (across all objects) is kept constant.

In our configuration of the model, we rely on two different recurrent phases. In one phase, the model runs with precise rollbacks. After a certain amount of simulation time, the model switches to approximated rollbacks. In this case, only the metadata describing the number of buffers kept in the simulation state are considered as core memory. Upon approximated state reconstruction, the model is parameterizable in the amount of missing state information which should be reconstructed so as to allow to continue the execution. This synthetic approach allows us to study what is the impact of the user-defined state reconstruction callback towards the overall execution time of the simulation model.

For our experiments we have used 100 total simulation objects. Given the platforms we have used, this is an interesting configuration. Indeed, the number of simulation objects per thread/core spans from 100 to 2 in the single-node bare-metal deploy, while it spans from 50 to 1 in the AWS deploy. Given the speculative nature of the ROOT-Sim runtime environment, this is a configuration with a high likelihood of local clocks' divergence among different simulation objects, leading to a non-minimal incidence of rollbacks.

The real-world application is an agent-based epidemiological Susceptible-Infected-Recovered (SIR) model [8] to study the spreading of diseases in large populations, with a focus on tuberculosis (TB). The model has been developed at the Barcelona Supercomputing Center (BSC) [9] to study the spreading phenomenon of this disease in the area around Barcelona. The model has been originally developed for being executed on top of the ROOT-Sim environment with no support for approximated rollbacks. We have slightly reorganized the application level code in order to embed the usage of the API that triggers the usage of the approximated rollback technique.

The model is based on agents (individuals) that circulate around an area, which is modeled by different simulation objects. Each object models a region in the target area, and the presence of agents in the region is recorder via information kept in the state of the simulation object modeling the region. In particular, the object keeps a record for each agent currently residing in the corresponding region, and the different agents model individuals who can be in one of five possible states according to TB infection dynamics: healthy, infected, sick (i.e., with active TB), under-treatment and recovered. The state variables of the individuals refer mainly to their status in the TB infection cycle as well as the time spent in such phases. Other individual parameters are age, native-immigrant origin, possible risk factors (e.g. smoking), and possible immunosuppression (mainly AIDS). Once a person is infected, the presence (or not) of pulmonary cavitation is also considered.

In our reshuffle of this model, done to exploit approximated rollbacks while speculatively running it, the individuals are implemented according to the Agent-based Modeling specification described in [13]. In this way, the agents belonging to different states are mapped to different hash tables, and only a subset of the hash tables belongs to the core memory. In particular, we have identified as core agents the ones in the healthy and recovered states. These agents are always restored accurately in the approximated rollback operation. The less relevant agents, namely those belonging to the infected, under-treatment, and sick states, are approximated. For them, the core only keeps the corresponding counter, but no individual-specific information (age, gender etc.) is actually maintained in the core, and restored in the approximated rollback. Loosing this information when the rollback occurs, and reconstructing it stochastically via the *CF* callback function, leads to deviating the characterizing features of the population (like the percentage of individuals within given age ranges, as well as the percentage of strangers and the percentage of individuals with other pathologies) corresponding to the values that were setup at startup of the simulation. In other words, an approximated rollback leads, for a given simulation run, to (slightly) change the sample of individuals moving around the area, which can clearly lead to changes in the statistics related to the spreading of the disease.

We have configured the epidemiological SIR model to carry out a micro-simulation of a medium metropolitan city. In particular, we have used 1600 simulation objects, each one covering a square region of 0,06 km^2, for a total simulated area of 96 km^2. The total number of agents managed by the model sums up to 1.6 millions. Agents move according to a random walk in the city area, so every simulation object manages on average 1,000 agents—a population density of 16,000 people per square kilometer. This configuration resembles the figures for the city of Barcelona in 2019. At simulation startup, 95.59% of the population is healthy, 4.28% of the population is infected, 0.12% of the population has recovered, and the remainder

0.01% is composed of sick or under-treatment people. Despite this very reduced number of contagious people, the agents are dynamic and frequently move, thus bringing the simulated environment into a pandemic quite quickly. We have simulated at total of 10 days of the SIR model evolution.

4.3 Results

We first discuss the results associated with the real-world application, namely the TB epidemic model. As mentioned in Section 3.3, our approximated rollback scheme can benefit even further from the joint exploitation of infrequent state saving of the core part of the object state. In Figure 6 we provide early performance results of the approximated rollback scheme compared to the traditional precise rollback scheme, when run in the single-node bare-metal deploy, with a periodic state saving interval set to $\chi = 10$ simulation events. This plot is essentially used to show the scalability in performance when increasing the computing power, which is similar for approximated and precise rollbacks—and to establish a baseline for moving in the configuration space of the log/restore support for a deep assessment. In fact, a single fixed value of the checkpoint interval is clearly not sufficient to characterize the potential of a state-rollback scheme, in terms of its effects on performance.

To better assess the validity of our approach, we have therefore run precise and approximated executions of the TB epidemic model while also varying the checkpoint interval χ. In particular, we have set the checkpoint interval χ to 1, 10, 20 and 40. To study and understand the behavior of the approximated rollback scheme, we have decided to compare against each other all the configurations which we have run. All the results associated with the TB epidemic model have been obtained by relying on the same sequence of pseudo-random numbers (i.e., the initial seed for the rollbackable random number generator has been explicitly and consistently set), and are averaged over 5 different runs.

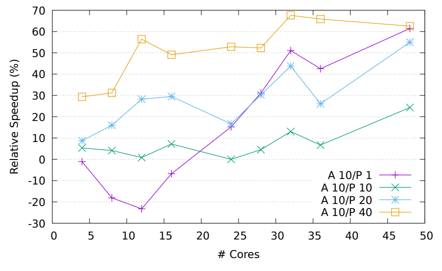
In Figures 7a–7c we report the relative performance of the approximated rollback scheme vs the precise rollback scheme, when comparing the different checkpoint intervals. The relative performance is computed as the difference between the total execution time of the precise and the approximated run, over the total execution time of the precise run. In all plots, we identify with the letter A the approximated scheme, and with P the precise scheme. The number after each letter is the corresponding checkpointing interval. By the results, we observe that in most situations we can benefit from an increased performance. The situations in which the traditional precise rollback scheme performs better than the approximated one (i.e., a negative relative speedup is observed) are associated with $\chi = 40$ for the approximated scheme, and χ set to lower values for the precise one. These simulation runs are characterized by a non-negligible amount of rollbacks, with a rollback length which is relatively short. Therefore, setting $\chi = 40$ for the approximated scheme requires to replay a non-minimal amount of events, which is otherwise not needed by the precise scheme. Hence, all the benefits associated with the reduced checkpoint/restore operation from the approximated scheme, which we will emphasize more with an additional set of experiments, are lost.

To prove that the approximated rollback technique is viable to provide useful (although potentially less precise) simulation results, in Figure 9 we report the actual simulation results from the model, run with $\chi = 10$. In particular, we plot the percentage of agents in each of the states (namely healthy, infected, sick, under-treatment, and recovered) for each simulated day. We recall that we store in the core memory of the simulation objects only the number of agents and the full information related to healthy and recovered agents. Despite the fact that we are also able to restore the number of infected, sick, and under-treatment agents, the inaccurate information related to each of these agents will produce a deviation in the simulation trajectory. By the results in Figure 9, we can observe that the trend in the number of agents in the different states observed across the simulation run for the case of approximated and precise rollbacks is comparable. Given the relative performance increase offered by the approximated rollback scheme, this result confirms that approximated rollbacks is a viable technique, especially if the simulation modeler is interested in trends.

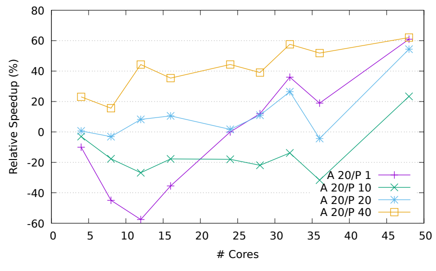
Our final experiment related to the TB epidemic model aims at understanding what is the effect of the state size in this model, with respect to the dynamics of the approximated rollback scheme. To this end, we have significantly scaled down the model, moving to a scenario in which each simulation object manages on average 20 agents, rather than the aforementioned 1,000. In this configuration, the impact of core vs non-core memory on the log/restore operation becomes less important. The results of this experiment, for the approximated scheme run with $\chi = 10$, are reported in Figure 10. As expected, we observe that we are able to provide a relative speedup which is less favorable when compared to that shown in Figure 7a.

To better understand the dynamics of the approximated rollback scheme, we have run a set of experiments relying on our variation of the PHold synthetic benchmark. In Figure 11 we report the execution time required to simulate 10K loop/buffer-migration events at each simulation object, on the single-node bare-metal deploy. We have varied the amount of state which is reconstructed by the CF restore function, spanning over 25%, 75%, and 100% of the simulation state. By the results, we can observe that the approximated rollback scheme provides a better performance with a higher core count (i.e., when the number of rollbacks is higher) in all state-reconstruction configurations. Also, the 100% state reconstruction configuration shows a better performance, due to the reduced cost for state saving which is observed with approximated rollbacks. With a small core count, the precise rollback scheme shows a performance which is better than the 75% state reconstruction configuration. We explain this phenomenon by the fact that the approximated scheme alters the sequence of operations on the critical path of the runtime environment, thus producing a slightly increased amount of rollbacks when running with a reduced amount of threads, yet with a limited amount of simulation objects.

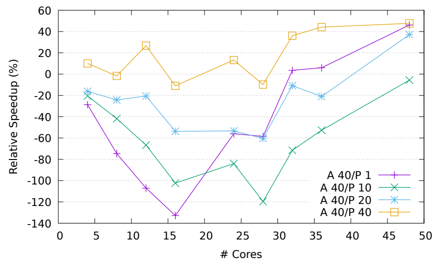
To complete our assessment using the PHold model when reconstructing 25% of the state, in Figures 8a, 8c, and 8b, we report the data for the AWS deploy related to the overall execution time, memory usage, and restore-operation latency, respectively. We note that, in this deploy, the intrusiveness of rollbacks is increased by the network latency. By the results, we can observe that approximated rollbacks can deliver a performance increase between 25% and 50% (Figure 8a). This aspect is clearly related to the time saved in the restore operation (Figure 8b) and the checkpoint operation.



(a) Approximated $\chi = 10$

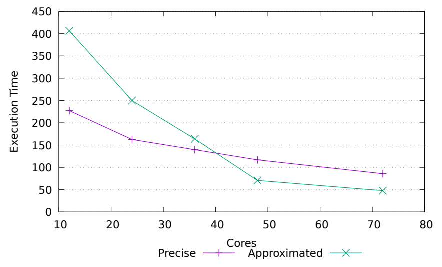


(b) Approximated $\chi = 20$

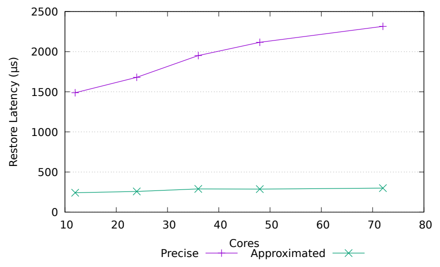


(c) Approximated $\chi = 40$

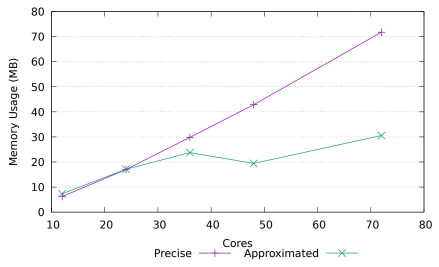
Figure 7: TB epidemic model - relative performance of the approximated vs precise rollback.



(a) Overall execution time.



(b) State restore latency.



(c) Memory footprint.

Figure 8: PHold results for the AWS deploy.

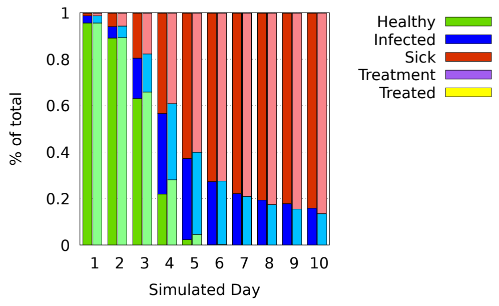


Figure 9: TB epidemic model - results accuracy of the approximated (lighter) vs precise (darker) rollback.

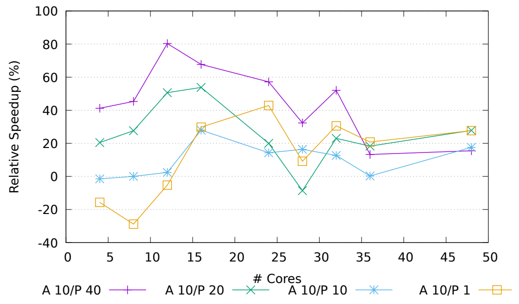


Figure 10: TB epidemic model - relative performance of the approximated ($\chi = 10$) vs precise rollback with scaled down workload.

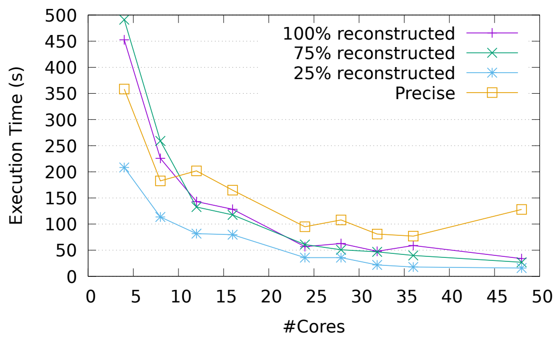


Figure 11: PHold - performance variation of the approximated rollback when reconstructing different amounts of the state (single-node bare-metal deploy).

Interestingly, a higher number of rollbacks produces the highest performance increase of 50%—as in the case of a large core count in Figure 8a. An additional benefit of the approximated rollback scheme is the fact that it further allows larger problems to become tractable when employing speculative processing schemes. Indeed, as shown in Figure 8c, the amount of memory required when relying on the approximated rollback scheme is significantly reduced.

5 CONCLUSIONS

We have presented approximated rollbacks, a technique which allows to save/restore only a subset of the state of a simulation object, relying on a user-defined state-reconstruction function to possibly guess the non-restored (missing) portions of the state. Our experimental assessment has shown that this approach can provide non-negligible performance improvements, both in local and distributed simulation environments. We have also assessed the impact of the approximated rollback technique on the statistical goodness of the outcome of the simulation, in particular for the case of an epidemic phenomenon of the tuberculosis disease. These results further show the viability of our proposal.

REFERENCES

[1] Christopher D Carothers, Kalyan S Perumalla, and Richard M Fujimoto. 1999. Efficient Optimistic Parallel Simulations Using Reverse Computation. *ACM Transactions on Modeling and Computer Simulation* 9, 3 (jul 1999), 224–253. https://doi.org/10.1145/347823.347828

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

[3] R.M. Fujimoto. 1999. Exploiting Temporal Uncertainty in Parallel and Distributed Simulations. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation.* IEEE Computer Society, 46–53. https://doi.org/10.1109/PADS.1999.766160

[4] Richard M Fujimoto. 1990. Parallel Discrete Event Simulation. *Commun. ACM* 33, 10 (oct 1990), 30–53. https://doi.org/10.1145/84537.84545

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

[6] Alonso Inostrosa-Psijas, Veronica Gil-Costa, Mauricio Marin, and Gabriel Wainer. 2018. Semi-asynchronous Approximate Parallel DEVS Simulation of Web Search Engines. *Concurrency and Computation: Practice and Experience* 30, 7 (apr 2018). https://doi.org/10.1002/cpe.4149

[7] David R Jefferson. 1985. Virtual Time. *ACM Transactions on Programming Languages and Systems* 7, 3 (jul 1985), 404–425. https://doi.org/10.1145/3916.3988

[8] William Ogilvy Kermak and Anderson Gray McKendrick. 1927. A Contribution to he Mathematical Theory of Epidemics. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* (1927). https://doi.org/10.1098/rspa.1927.0118

[9] Cristina Montañola-Sales, Joan Francesc Gilabert-Navarro, Josep Casanovas-Garcia, Clara Prats, Daniel Lopez, Joaquim Valls, Pere Joan Cardona, and Cristina Vilaplana. 2015. Modeling Tuberculosis in Barcelona. A Solution to Speed-up Agent-based Simulations. In *Proceedings of the 2015 Winter Simulation Conference (WSC).* IEEE, 1295–1306. https://doi.org/10.1109/WSC.2015.7408254

[10] Alessandro Pellegrini and Francesco Quaglia. 2014. The ROme OpTimistic Simulator: A tutorial. In *Proceedings of the Euro-Par 2013: Parallel Processing Workshops.* LNCS, Springer-Verlag, 501–512. https://doi.org/10.1007/978-3-642-54420-0_49

[11] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2009. Di-DyMeLoR: Logging only Dirty Chunks for Efficient Management of Dynamic Memory Based Optimistic Simulation Objects. In *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS).* IEEE, 45–53. https://doi.org/10.1109/PADS.2009.24

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

[13] Andrea Piccione, Matteo Principe, Alessandro Pellegrini, and Francesco Quaglia. 2019. An Agent-Based Simulation API for Speculative PDES Runtime Environments. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS).* ACM Press, New York, New York, USA, 83–94. https://doi.org/10.1145/3316480.3322890

[14] Bruno R Preiss, Wayne M Loucks, and Ian D. Macintyre. 1994. Effects of the Checkpoint Interval on Time and Space in Time Warp. *ACM Transactions on Modeling and Computer Simulation* 4, 3 (jul 1994), 223–253. https://doi.org/10.1145/189443.189444

[15] Francesco Quaglia. 2001. A Cost Model for Selecting Checkpoint Positions in Time Warp Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems* 12, 4 (apr 2001), 346–362. https://doi.org/10.1109/71.920586

[16] Francesco Quaglia and Roberto Beraldi. 2004. Space Uncertain Simulation Events: Some Concepts and an Application to Optimistic Synchronization. In *18th Workshop on Parallel and Distributed Simulation, 2004. PADS 2004.* IEEE, 181–188. https://doi.org/10.1109/PADS.2004.1301299

[17] D.M. Rao, N.V. Thondugulam, R Radhakrishnan, and P.A. Wilsey. 1998. Unsynchronized Parallel Discrete Event Simulation. In *Proceedings of the 1998 Winter Simulation Conference.* WSC, Vol. 2. IEEE, 1563–1570. https://doi.org/10.1109/WSC.1998.746030

[18] R. Ronngren, M Liljenstam, J Montagnat, and Rassul Ayani. 1996. Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS).* IEEE, 70–77. https://doi.org/10.1109/PADS.1996.761564

[19] H.M. Soliman and A.S. Elmaghraby. 1998. An Analytical Model for Hybrid Checkpointing in Time Warp Distributed Simulation. *IEEE Transactions on Parallel and Distributed Systems* 9, 10 (1998), 947–951. https://doi.org/10.1109/71.730524

[20] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. 2009. Benchmarking Memory Management Capabilities within ROOT-Sim. In *Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT).* IEEE, 33–40. https://doi.org/10.1109/DS-RT.2009.15

[21] Darrin West and Kiran Panesar. 1996. Automatic Incremental State Saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS).* IEEE, 78–85. https://doi.org/10.1109/PADS.1996.761565