Mixing Hardware and Software Reversibility for Speculative Parallel Discrete Event Simulation

Davide Cingolani, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia

Sapienza, University of Rome

Abstract. Speculative parallel discrete event simulation requires a support for reversing processed events, also called state recovery, in case they reveal as causally inconsistent. In this article we present an approach where state recovery relies on a mix of hardware- and software-based techniques. Particularly, we exploit the Hardware Transactional Memory (HTM) support, as offered by Intel Haswell CPUs, to process events by the application code as in-memory transactions, which are possibly committed only after their causal consistency is verified. At the same time, we exploit an innovative software-based reversibility technique, fully relying on transparent software instrumentation targeting x86/ELF objects, which enables undoing side effects by events with no actual backward recomputation. Each thread within our multi-thread speculative processing engine dynamically (namely, on a per-event basis) selects which recovery mode to rely on (hardware vs software) depending on varying runtime dynamics. The latter are captured by a lightweight model indicating to what extent the HTM support (not paying any instrumentation cost) is efficient, and after what level of events' parallelism it starts degrading its performance, e.g., due to excessive data conflicts while manipulating causality meta-data within HTM based transactions. We released our implementation as open source software and provide some experimental results for an assessment of its effectiveness.

1 Introduction

When dealing with Discrete Event Simulation (DES), its move onto parallel architectures has been historically based on the Parallel Discrete Event Simulation (PDES) paradigm [7]. In this kind of simulation, as well as in the traditional DES paradigm, the evolution of the system is described in terms of *timestamped discrete events*, which are impulsive—they happen at a specific simulation time instant, the timestamp of the event, and have no duration. Parallelism is achieved in PDES by partitioning the simulation model into several distinct entities, called *simulation objects* or *logical processes* (LPs). Each LP is associated with a private simulation state—the whole simulation state is the union of these private states—and the execution of an impulsive simulation event at any LP produces a state transition on the state of the LP itself. The privateness of the LPs' simulation states implies that information exchange across different LP is only supported via the exchange of events, which can be generated (in any number) during the execution of whichever event.

PDES speculative execution [10] allows processing events with no previous assurance of their causal consistency. This means that an event destined to some LP can be dispatched for execution with no guarantee at all regarding the fact that no other events with a higher priority, say lower timestamp, will be ever received by that same LP in the future. Such events, referred to as *straggler events*, are the a-posteriori materialization of a timestamp-order violation, also referred to as *causal violation*. Such violations require some state recovery (reversibility) support for undoing the side effects on the LPs' states which are associated with inconsistent processing of events.

In literature, the reversibility support has been traditionally based on pure software implementations exploiting either checkpointing techniques (see, e.g., [15, 16]) or reverse computing ones (see, e.g., [2]). A few other approaches have been based on off-loading the checkpoint task to off-the-shelf or unconventional hardware [9, 18]. More recently, the Hardware Transactional Memory (HTM) support offered by modern processors, such as the Intel Haswell, has been taken into consideration in order to enable the speculative execution of events as in-memory transactions [19], making them automatically recoverable with low overhead thanks to the reliance on the hardware transactional cache. However, to the best of our knowledge, there has been no attempt to exploit hardware and software based reversibility in a synergic combination for speculative PDES.

In this article we present a speculative PDES engine, oriented to multi-core machines, which is based on such a kind of hardware/software combination. Particularly, we enable each concurrent worker thread operating within the engine to dynamically select the best suited reversibility support among two: (1) one relying on HTM facilities inspired to [19] and (2) another relying based on software reversibility, particularly in the form of *undo code blocks* [3]. The dynamic selection is based on the consideration that not all the speculatively executed events are *valuable* in the same manner when run as HTM transactions due to several reasons. A first one deals with the fact that the final commit of the transaction needs to check/update causality meta-data, hence the higher the degree of concurrency while accessing these meta-data, the higher the likelihood of yielding to data conflicts that lead to the abort of the HTM transactions. Also, causality meta-data are updated according to the progress of the commit horizon of the PDES run, as determined along time by the commit of the event with the lowest timestamp. Hence speculatively processed events with HTM support that are further ahead of the commit horizon will need to find causality meta-data reflecting more updates upon trying to commit, which again leads to an abort if these updates were not yet issued by the commitment of events with higher priority, say lower timestamps. Finally, the HTM support is limited to transactions whose read/write set fits (with no capacity conflict by other cores of the same CPU) the transactional hardware cache. Hence for models with events that (or execution phases where the events) have large data sets the likelihood of successfully committing the corresponding HTM transactions may be (significantly) reduced.

We overcome these drawbacks in our speculative PDES engine by dynamically enabling any worker thread to process an event not as an HTM transaction (just to reduce the likelihood of running non-valuable transactions), but rather via a modified version of the original event-handler code. This version is transparently instrumented in order to be able to generate (at runtime) the minimal set of machine instructions (the so called undo code block) that allows reversing any memory side effect. In the instrumentation process we target x86/ELF objects. The possibility to commit events run with software reversibility is no longer bound to the possibility to commit an HTM transaction. This leads to the scenario where the engine is able to improve fruitful usage of computing resources just because of the possibility to exploit the HTM support in the most valuable manner, while jointly relying on a bit more costly software reversibility when valuable hardware based reversibility would be impaired.

Clearly, the coexistence of HTM and software based reversibility (with concurrent threads relying on one or the other at a given time instant) needs solutions in order to avoid that the two techniques do not interfere with each other. Specifically, valuable HTM work should not be interfered by software reversibility based one. For the case of concurrent speculatively processed events bound to the same LP (hence operating within the same local state) this is achieved by introducing a prioritization mechanism that leads an HTM processed event to gain higher priority with respect to the events processed with the software reversibility support. So the latter will never concurrently access (any portion of) the overall data set—say LP state as a whole—possibly targeted by the HTM transaction, hence not leading to its abort. On the other hand, we still enable inter-LP concurrency, thus enabling the so called weak-causality model [17], by not preventing multiple HTM transactions to successfully operate on disjoint data sets within the LP state. Also, given that in our software reversibility scheme we avoid the usage of checkpointing (in fact the undo code block is not a log of data, rather of machine instructions), we avoid at all the typically large usage of memory by checkpointing (only partially resolved by incremental checkpointing schemes) hence further reducing the (potential) problems related to limited cache capacity issues of the HTM support and conflicting cache accesses by the threads.

Our engine has been released as open source software[1], and we also provide some experimental data for an assessment of its effectiveness when running the classical Phold PDES benchmark [8] on an Intel Haswell processor, with HTM support, equipped with 4 physical cores.

The remainder of this article is structured as follows. In Section 2 we discuss related work. In Section 3 we present the methodology standing behind hardware- and software-reversibility based execution of PDES models, and we describe the design principles characterizing our mixed simulation engine architecture. Section 4 presents an experimental assessment of our proposal.

[1] `https://github.com/HPDCS/htmPDES/tree/reverse`

2 Related Work

The state restore operation is of fundamental importance in speculative PDES, and has therefore been extensively studied in the literature. Two main incarnations of state restore schemes have been proposed, one based on *state checkpoint and reload*, and one based on *reverse computing*. The former flavour is based on the possibility for the simulation engine to know what are the memory buffers that keep each LP's simulation state, which are copied onto a separate buffer—called the *simulation snapshot*—at a given point of the execution. In this way, undoing a chain of wrongly-computed events (namely, state updates) boils down to selecting a simulation snapshot which is still consistent (i.e., it was taken at a simulation time smaller than the straggler's one). This snapshot is then copied onto the LP live state image, thus undoing the effects of causal-inconsistent events. This approach is both memory- and computationally-intensive, and might lead to poor simulation performance, since if no causal inconsistency is detected at all, resources are spent for taking unnecessary snapshots. To this end, several proposals have addressed the possibility to take state snapshots less frequently (see, e.g., [16]) or in an incremental way (see, e.g., [21]) or combining the two schemes (see, e.g., [15, 20]). Other solutions rely on hardware support to offload from the CPU the memory copy for taking the checkpoint. Specifically, the work in [18] proposed to exploit programmable DMA engines to perform the copy, while [9] presents the design of a so called rollback-chip, a hardware facility that automatically saves old versions of state variables upon their updates. Both these approaches, reduce the CPU-time for checkpointing tasks but do not directly cope with memory usage.

Reverse computing is instead based on the notion of *reverse events*. A reverse event $\bar{e}$ associated with a forward event e is an event such that if the execution of e produces the state transition $e(S) \rightarrow S'$, the execution of $\bar{e}$ on S' produces the inverse transition $\bar{e}(S') \rightarrow S$. Such reverse events could be implemented manually [2] or via compiler-assisted approaches [12]. Although reverse computation is much less memory-greedy than checkpointing, the main issue with this approach lies in the *rollback length*, namely the number of events which must be undone upon a state restore operation. In particular, the total cost of a rollback operation is directly proportional to the number of undone events and their granularity, as reverse events re-process (although in a reversed fashion) all the steps of a forward event, even if some of them are not directly related to state updates.

The more recent proposal in [3] has tackled the state restore operation via software reversibility through the adoption of *undo code blocks*. The goal of this approach is to reduce the time-complexity of the rollback operation, making the reversibility of events independent of the forward execution's granularity. This is done by relying on static binary instrumentation, targeting x86-64/ELF objects, where the simulation model's code is scanned searching for all machine-level instructions which entail a memory update. These instructions are transparently augmented with an ad-hoc routine which computes the target address of the memory write just before it takes place, so that the original value is directly

packed into an on-the-fly assembled machine instruction whose execution restores it. All these runtime generated assembly instructions are stored into an undo code block which, when executed, undoes all the effects of the execution of a forward event on the simulation state. This solution finds a good balance between incremental checkpointing—no actual meta-data are required to restore a previous state—and reverse computing—the execution cost of an event is no longer dependent on the complexity of forward events. Nevertheless, if an event is unlikely to be undone due to a rollback operation, the cost of tracing memory updates and generating undo code block is paid unnecessarily.

Another recent proposal [19] exploits HTM facilities offered by modern Intel Haswell CPUs to allow running simulation events within transactions. An ad-hoc routine determines whether the execution of an event is safe or not, by checking compact shared meta-data keeping track of the simulation time associated with the events that are being run by the concurrent threads. The event associated with the smallest timestamp is considered safe, and it is therefore the only event which is executed outside of a transaction. By using this scheme, all the events which are transactionally executed are automatically aborted if a conflict on the same data structures is detected. At the end of a transaction, the safety of the just-executed event is evaluated again, and in case the event has become safe, it is then committed. In the negative case, the transaction is immediately aborted and (possibly) restarted, because the access to the shared meta-data makes it doomed if the event is not safe yet—in fact, another thread will eventually update the content of the meta-data, to indicate that the execution of a safe event has been completed. A dynamic throttling strategy is used to increase the likelihood of committing a transaction, by delaying the time instant at which the shared meta-data are accessed.

Our work differs from previously published work since none of the aforementioned proposals makes use of a combination of hardware and software reversibility for state restore operations. Particularly, we use the results in [19] and [3] as baselines for building a mixed hardware/software recoverability support that takes the advantages of the two different techniques As pointed out in the introduction, we dynamically resort to undo code blocks (thus paying the cost of running an instrumented code version) only in case valuable speculative work cannot be carried out (by a thread at some point in time) via the reliance on HTM. Thus we pay the overhead of software reversibility only when HTM based reversibility does not pay off (or is inviable due to, e.g., transactional cache capacity limitations).

3 The Hardware/Software Reversibility Based Engine

3.1 Basics

We target a baseline speculative PDES engine structure that is independent of the actual reversibility support, whose schematization is provided in Figure 1. In compliance with traditional PDES, the engine supports the partitioning of the simulation model into n distinct LPs, each one associated with a unique ID in the

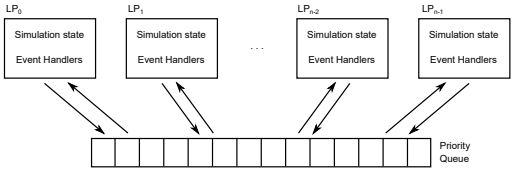

Fig. 1. Basic engine organization.

range $[0, n-1]$. Each LP is associated with a private simulation state (although possibly scattered on dynamic memory) and with one or more event handlers representing the code blocks in charge of processing the simulation events and generating state updates, as well as of (possibly) producing new events to be injected in the system. The delivery of a simulation event to the correct handler is demanded from the underlying simulation kernel, which is also in charge of guaranteing consistency of a shared event pool that keeps all the already scheduled events, as well as causal consistency of the updates occurring on the LPs' states. Concerning the event pool, we rely on a shared lock-protected global queue, particularly a calendar queue [1]. Multiple concurrent worker threads can extract events from the event pool and can concurrently dispatch the execution of the corresponding LPs by activating some event handler as a callback function.

3.2 Simulation Horizons and Valuability of Speculative Work

In speculative PDES, we can always identify a point on the simulation time axis which is the *commit horizon*—commonly referred to as Global Virtual Time (GVT). This is the simulation time instant that distinguishes between events which might be undone (e.g., due to some causality violation) and events which will never be undone. This time instant can be logically identified by considering that any simulation event e executed at simulation time T can only generate some new event e' associated with timestamp $T' \geq T$. In fact, violating this assumption would imply that an event in the future might affect the past, which is clearly a non-meaningful condition for any real-world process/phenomenon. Therefore, to identify the commit horizon, it is sufficient to identify, across all the events which are currently scheduled at (or have just been processed by) any LP in the system the one associated with the minimum timestamp. Such timestamp corresponds to the commit horizon. In fact, no event still to be executed in the system might produce a causal inconsistency involving the LP in charge of the execution of the commit horizon event[2].

With our target engine organization, the commit horizon is associated with the oldest event that is currently being executed (or has just been executed) at any worker thread. Therefore, keeping track of the commit horizon boils down to registering, for each worker thread, the timestamp of the event e currently

[2] Simultaneous events do not violate this assumption. Nevertheless, if not properly handled by some tie-breaking function [11, 13], they could induce livelocks in the speculative execution.

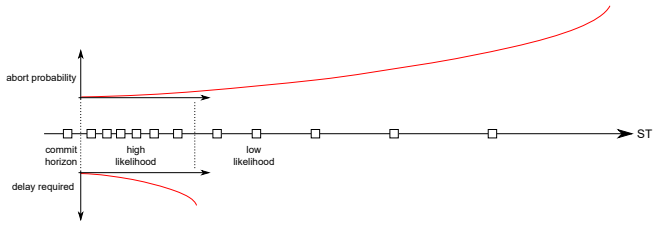

Fig. 2. Three logical regions on the simulation time axis, with varying density of pending events—those still to be processed, which will possibly generate new ones along the simulation time (ST) axis.

being executed, by replacing the value only after a new event is fetched for processing from the event pool, so that any new event possibly produced by e has its timestamp already reflected into the event pool. The commit horizon can be computed as the minimum among the registered values.

At any time, the commit horizon event can be considered as a *safe* (namely, causally consistent) one, and therefore does not require any reversibility mechanism for its execution. Let us now discuss about the likelihood of safety of other events to be processed, which stand ahead of the commit horizon. Empirical evidence plus statistical considerations based on classical distributions for the timestamp increment driving the generation of events in common simulation models (see, e.g., [5, 6]) have shown that event patterns are, at any time, characterized by greater density of events, say locality of activities, in the near future of the actual GVT. This situation is depicted in Figure 2. Also, such locality tends to move along the time axis just based on the advancement of the commit horizon. The implication is that the risk of materialization of causal inconsistencies when speculatively processing one event that is ahead of the commit horizon is somehow linked to its distance from such horizon. This is also linked to the notion of lookahead of DES models, a quantity expressing the minimal timestamp increment we can experience for a given model when processing whichever event that originates new events to be injected in the system. Larger lookahead leads to produce new events in the far future, hence those getting closer to (although not coinciding with) the current commit horizon become automatically safe.

By this consideration, the speculative processing of events that are closer to the commit horizon looks more valuable in terms of avoidance of causality inconsistencies, hence our approach is to enable the processing of these events as HTM-based transactions, say via the more efficient (lower overhead) recoverability support. We also note that running events that are close to the commit horizon as HTM-based transactions will also lead to faster advancement of such horizon, as compared to what we would expect if running them via software-based reversibility, since this would lead to longer processing times due to the overhead for producing the undo code blocks. However, an HTM-based transaction can commit only after events standing in the past have already been committed and the corresponding worker threads have already updated their entries in the meta-data array keeping their current timestamp. So, in order to

increase the likelihood of committing the HTM-based transactional execution of some event, this transaction typically needs to include a busy-loop delay enabling a wait phase just before checking whether the meta-data were updated[3]. Checking the meta-data at some wrong point in time will in its turn lead to the impossibility to recheck these data fruitfully in the future, since the updates occurring between the two checks will lead to a data conflict and to the abort of the checking transaction. In Figure 2 we show how such a delay should be selected somehow proportionally to the distance (in terms of event count) of the event processed via HTM support from the commit horizon. Overall, for events that are further ahead from the commit horizon, the delay could not pay off, hence a more profitable approach to speculatively processing them is the one to run them outside the HTM-based transaction, still with reversibility guarantee achieved via software.

The problem of determining what is the threshold distance from the commit horizon beyond which HTM support does not pay off is clearly also related to the interference between concurrent HTM-based transactions when using the underlying hardware resources. In fact, if we experience a scenario where two concurrent transactions both require large transactional cache storage for executing the corresponding dispatched events, and the cache is shared across the cores, then even if an event would ideally reveal as causally consistent upon attempting to finalize the transactions, it would anyhow be doomed to abort due to cache capacity conflicts. A similar cache capacity-due abort may even be experienced in case of single HTM-based transaction instance, just depending on the transaction data set, which might exceed the cache capacity.

To cope with the runtime adaptive selection of the threshold value, we rely on a hill climbing scheme based on the following parameters, easily measurable at runtime across successive wall-clock-time windows:

- T_{HTM}, the total processing time spent across all the worker threads while processing events (either committed or aborted) via HTM support
- $COMMIT_{HTM}$, the total number of committed events whose speculative execution has been based on HTM support;
- T_{soft}, the total processing time spent across all the worker threads while processing events (either committed or aborted) that are made recoverable via software-based support (here we include the time spent for instrumentation code used to generate undo code blocks, plus the time for running the undo code blocks in case the events are eventually undone);
- $COMMIT_{soft}$, the total number of committed events whose execution has been based on the software support for recoverability.

By the above quantities, we compute the so called work-value ratio (WVR) for both HTM-based and software-based recoverability just like:

$$WVR_{HTM} = \frac{T_{HTM}}{COMMIT_{HTM}} \qquad WVR_{soft} = \frac{T_{soft}}{COMMIT_{soft}} \qquad (1)$$

[3] Other kind of delays, such as operating system sleeps, are unfeasible since any user/kernel transition will lead an HTM-based transaction to abort deterministically on current HTM-equipped processors.

which express the average amount of CPU time required for performing useful work (namely, for processing an event that is not undone) with the two different recoverability supports. Then, the threshold value THR determining the commit horizon distance (evaluated as event count) beyond which we consider it more convenient to process the event via software reversibility, rather than HTM-based one, is increased or decreased depending on whether the relation $WVR_{HTM} \leq WVR_{soft}$ is verified (as computed on the basis of statistics, on the baseline parameters listed above, collected in the last observation window). In order to avoid stalling in local minima (e.g. due to the avoidance of runtime samples for any of the above listed parameters), we intentionally perturb THR by ± 1 within the hill climbing scheme if its value reaches either zero or the number of threads currently running in the PDES platform.

3.3 Engine Architecture

As mentioned, our engine allows the co-existence of hardware-based and software-based reversibility facilities. While introducing hardware-based reversibility facilities is somehow easy—it can be done using the primitives `TRANSACTION_START`, `TRANSACTION_END`, and `TRANSACTION_ABORT` to drive event processing—software-based reversibility requires a bit more care, especially when targeting full transparency to the application-level developer. To cope with this issue, we rely on *static binary instrumentation*. In particular, we exploit the Hijacker [14] open-source customizable static binary instrumentation tool. Using this tool, we are able (before the final linking stage of the application-level simulation model) to identify any memory writing instruction (either a simple `mov` or a more complex ones, like `cmove` or `movs` instructions) and to place just before each memory-update instruction a call to a `reverse_generator` module which reads the current value of the target memory location so as to directly generate the reverse instruction able to undo the corresponding side effect according to the proposal in [3]. The sequence of reversing instructions for a same event forms the undo code block of the event. Clearly, the instrumented and the non-instrumented versions of the application modules also need to coexist (since the non-instrumented version is the one to be run in case of HTM-based reversibility). Such coexistence has been achieved by using a multi-coding scheme when rewriting the ELF of the program at instrumentation time, and by identifying the entry points to the two versions of code (instrumented and not) within the same executable using function pointers exposed to the PDES engine.

In our implementation the reversing instructions associated with an event (those forming the undo code block of the event) are organized into a *reverse window*, which is used as a stack of negative instructions that can be invoked via a `call`. Correct execution of an undo code block is ensured by the presence of a `ret` instruction at the end of the reverse window. Also, if the forward execution of an event updates multiple times the same memory location, only the first instruction updating that location should be associated with the generation of an inverse instruction, since the following updates would be anyhow undone by

Algorithm 1 Shared Lock Acquisition/Release

```
 1: int lock_vector[n]
 2: double timestamp[n]                                   ▷ To avoid priority inversion
 3: int thread_id[n]                                      ▷ To avoid priority inversion
 4: procedure LOCK_LP(e, LP, mode, locking)
 5:     if mode = EXCLUSIVE then
 6:         acquired ← false
 7:         while ¬acquired ∧ locking do
 8:             while lock_vector[LP] > 0 do
 9:                 nop
10:             old_lock ← lock_vector[LP]
11:             if CAS(-1, old_lock, lock_vector[LP]) then
12:                 acquired ← true
13:     else
14:         acquired ← false
15:         while ¬acquired ∧ locking do
16:             while lock_vector[LP] < 0 do
17:                 nop
18:             old_lock ← lock_vector[LP]
19:             if CAS(old_lock + 1, old_lock, lock_vector[LP]) then
20:                 acquired ← true
21:     if ¬acquired then
22:         atomically {
23:             timestamp[LP] ← T(e)
24:             thread_id[LP] ← thread_id
25:         }
26:     return acquired
27: procedure UNLOCK_LP(LP, mode)
28:     if mode = EXCLUSIVE then
29:         lock_vector[LP] ← 0
30:     else
31:         do
32:             old_lock ← lock_vector[LP]
33:         while ¬ CAS(old_lock − 1, old_lock, lock_vector[LP])
```

the first inverse instruction. We therefore employ a fast hashmap to keep track of destination addresses within a forward event. Whenever `reverse_generator` is activated, this hashmap is queried to determine whether the destination address was already involved in a negative instruction generation.

As mentioned before, to ensure consistency and minimize the effects of data contention on HTM-based execution of events, we must ensure that at no time two different worker threads can execute both software-reversible and hardware-reversible events at once, which target the same LP state. In fact, if this would happen, we might incur the risk of having less valuable work to invalidate more valuable one (since the HTM-based transaction would be aborted if its data set would overlap the write set of the event executed via software-based reversibility). Also, we cannot allow two (or more) events run via software-based reversibility to simultaneously target the same LP state. In fact, these events would not be regulated by any transactional execution scheme[4]. To this end, we rely on a synchronization mechanism similar in spirit to an atomic shared read/write lock [4]. Whenever a worker thread extracts an event from the shared

[4] The undo code blocks guarantee reversibility of memory updates limited to events executing the updates on the LP state in isolation, which complies with classical PDES where each LP is an intrinsically sequential entity.

event pool, it first determines whether it should execute it using hardware-based or software-based reversibility according to the policy introduced in Section 3.2. If the selected execution mode is HTM-based, the worker thread tries to acquire the lock on the target LP in a non-exclusive way, which nevertheless fails (i.e., requires spinning) in case any other worker thread already took it in an exclusive way. On the other hand, if the selected execution mode is based on software reversibility, the worker thread tries to acquire the lock in an exclusive way, yet this operation requires spinning if at least one worker thread has non-exclusively taken the lock. Nevertheless, this approach might lead to some priority inversion, among the threads which are running more valuable events via HTM support and threads which are running less valuable events via software-based reversibility. To avoid this, we use a *locking* flag to instruct the algorithm to avoid spinning if it was not possible, for any reason, to acquire the lock—namely, setting *locking* to false transforms the lock into a trylock. Therefore, if the lock is not taken, two additional values in two arrays are updated atomically: `timestamp` and `thread_id`, which are exploited on a per-LP basis. In particular, the worker thread registers the timestamp it has an event to process at, and its thread id. The latter value is only used to create a total order among threads in case simultaneous events are present, to avoid possible deadlock conditions. These values are periodically inspected by other worker threads (upon a safety check for the current processed event, which fails), so as to determine whether some higher priority event is waiting. In that case, if the work carried out is not likely to be committed shortly, thanks to the reversibility supports it gets squashed, so that higher priority is given immediately to events at a smaller timestamp. Algorithm 1 shows the lock management pseudo-code, which relies on the Compare and Swap (CAS) read-modify-write primitive to increase/decrease the value of a shared per-LP counter. Value -1 for the counter means that the lock is exclusively taken, while value 0 indicates that no thread is running an event bound to the LP. A positive value is a sort of reference counter which tells how many worker threads are concurrently executing events via hardware-based reversibility.

We can now discuss the organization of the main loop of threads within our speculative PDES engine, whose pseudo-code is shown in Algorithm 2. Essentially, it is made up by three different execution paths, each one associated with one of the different execution modes. Initially, a call to a FETCH() procedure allows to extract from the shared event pool the event with the smallest timestamp. Then, a statistical approximation of the number of events which are expected to fall before the currently fetched event (since others my still be processed or might be produced as a result of the processing) is computed as:

$$\frac{T(e) - commit\_horizon}{average\_timestamp\_increment} \tag{2}$$

where $average\_timestamp\_increment$ is computed as $\frac{commit\_horizon}{total\_committed\_events}$ ([5]). This value, together with the threshold THR (see Section 3.2), is used to

[5] For non stationary models, where the distribution of the timestamp increment between successive events can change over time in non-negligible way, this same statis-

Algorithm 2 Main loop

```
1:  procedure MainLoop
2:      new_events = ∅                          ▷ Set of events generated during the execution of an event
3:      while ¬endSimulation do
4:          e ← Fetch()
5:          if e = NULL then
6:              goto 3
7:          events_before ← (T(e) − commit_horizon) / average_timestamp_increment
8:          if Safe() then                                  ▷ Safe execution: on the commit horizon
9:              Lock_LP((e, LP(e), NON_EXCLUSIVE, true))
10:             new_events ← ProcessEvent(e)
11:             Unlock_LP(LP(e), NON_EXCLUSIVE)
12:         else if events_before ≤ THR then        ▷ HTM-based execution: high likelihood region
13:             if ¬ Lock_LP((e, LP(e), NON_EXCLUSIVE, false)) then
14:                 goto 7
15:             BeginTransaction( )
16:             new_events ← ProcessEvent(e)
17:             Throttle(events_before)
18:             if Safe() then
19:                 CommitTransaction( )
20:                 Unlock_LP(LP(e), NON_EXCLUSIVE)
21:             else
22:                 AbortTransaction( )
23:                 Unlock_LP(LP(e), NON_EXCLUSIVE)
24:                 goto 7
25:         else                                    ▷ Software-reversible execution: low likelihood region
26:             if ¬ Lock_LP((e, LP(e), EXCLUSIVE, false)) then
27:                 goto 7
28:             SetupUndoCodeBlock( )
29:             new_events ← ProcessEvent_Reversible(e)
30:             while ¬ Safe( ) do
31:                 if timestamp[LP] < T(e) ∨ ( timestamp[LP] = T(e)∧ thread_id[LP] < tid) then
32:                     Unlock_LP(LP(e), EXCLUSIVE)
33:                     UndoEvent(e)
34:                     new_events = ∅
35:                     goto 7
36:         Flush(e, new_events)
37:         atomically {
38:             if thread_id[LP] = tid
39:               timestamp[LP] ← T(e)
40:               thread_id[LP] ← tid
41:         }
```

determine whether a certain event might be more valuable or not, thus requiring either HTM-support or software-based reversibility (line 12). Additionally, it an event is executed exploiting HTM, this value drives as well the selection of a delay before checking again the safety of the corresponding transaction (namely, whether the timestamp of the event has in the meanwile become the commit horizon), so as to avoid making it doomed with a high likelihood (line 17).

In case of a safe execution, i.e. the execution of the event on the commit horizon (lines 8–11), we take a non-exclusive lock, which is used to inform any other thread that the destination LP is currently processing an event. This avoids that any other worker thread starts processing an event via software-based reversibil-

tic could be simply rejuvenated periodically, by discarding non-recent events commitments and subtracting from *commit_horizon* the upper limit of the discarded simulation time portion.

ity at the same LP while we are processing in safe mode. Moreover, we configure the lock to spin because the worker thread in charge of executing this event has the highest priority and any other competing thread will try to give it permission to continue execution as fast as possible.

For a transactional execution (lines 12-24), we use the trylock version of the per-LP lock. If we fail to acquire the lock, the execution resumes from line 7, meaning that we check again whether the extracted event has become safe or not, in the meanwhile. Otherwise, as already explained before, we start executing the event within an HTM-based transaction, introducing an artificial delay—via the THROTTLE($events\_before$) call—which is proportional to the estimated number of events in between the commit horizon and the currently executed event. If the transaction becomes doomed (lines 21–24) the execution restarts from line 7, so as to check whether the just-aborted event has become safe.

The case of execution via software reversibility (lines 25–34) is a bit different. In fact, first we have to take an exclusive lock—in a trylock fashion, for the same consideration related to the HTM execution—and we have to setup the undo code block, by allocating a reverse window buffer. At the end of the execution of the event, similarly to the HTM-based case, we have to wait for the event to become safe. Nevertheless, since this execution entails taking an exclusive lock, we continuously check whether some other thread is registered at the same LP with a higher priority (line 31). This situation might arise due to another event, executed at any other worker thread, generating a new event to the same LP with a timestamp smaller than the one of the event currently processed via software-based reversibility. Failing to make this specific check could either hamper liveness (a thread waits its event to be the commit horizon, which cannot happen) or correctness (events are committed out of order). Line 31, paired with lines 21–25 of Algorithm 1, is able to ensure both correctness and liveness.

Whenever an event is executed, and then committed thanks to safety assurance, in whichever execution mode, we first place into the calendar queue any possible new event generated (line 36), and we then unregister the thread from the `timestamp` and `thread_id` vectors which are used to avoid priority inversion (lines 37–40). For the implementations of FETCH(), FLUSH(), and SAFE(), we refer the reader to [19].

4 Experimental Results

We tested our proposal with the Phold benchmark for PDES systems [8]. This is made up by synthetic LPs whose behavior can be tailored depending on test scenario one would like to generate. We included 1024 LPs in the simulation model, each one scheduling events for itself or for the other objects. Specifically, upon processing an event, the probability to schedule a new event destined to another simulation object has been set to 0.2, which is representative of scenarios with non-minimal interactions across the simulated parts. Also, the initial population of events has been set to 1 event per simulation object, while the timestamp increment determining the actual timestamp of newly scheduled events has been

set to follow the exponential distribution with mean value equal to one simulation time unit. The model lookahead has been set to a minimal value computed as the 0.5% of the average timestamp increment. Further, the overall simulation is partitioned in to 4 phases where the LPs exhibit alternate behaviors in terms of updates into their states. Specifically, phases 1 and 3 are write-mild since each event only updates the classical counter of processed events and a few other statistical values within the LP state. Contrariwise, phases 2 and 4 are write-intensive, since event processing also updates an array of counters' values, still embedded with the LP state (particularly, by performing 500 updates on the array entries). Overall, the different phases mimic varying locality and memory access profiles one might expect from real applications' workloads. A classical busy-loop characterizing PHOLD event processing steps is also added which is set to generate an average event granularity of about 25 microseconds. In this experiment, we compare the performance of our mixed hardware- and software-based approach to both pure hardware-based reversibility (as proposed in [19]) and pure software-based one exclusively relying on undo code blocks (this is achieved by preventing any thread to exploit HTM in our engine). We did not compare with the performance achievable by some last generation traditional speculative PDES platform just because the data reported in [19] have shown that event granularity values of a few (tens of) microseconds do not allow this type of platforms to provide significative speedup values (due to the fact that they are based on explicit partitioning of the workload across the threads, and on explicit message passing for event cross-scheduling, thus resulting much more adequate for larger grain simulation models). Overall, we assessed our proposal with a workload configuration just requiring alternative forms of speculative parallelization (like the one we propose), as compared to the classical ones.

We have run this benchmark by varying the number of employed threads from 1 to the maximum number of physical CPU-cores in the underlying HTM-equipped machine, which is equipped with two Intel Haswell 3.5 GHz processors, 24 GB of RAM and runs Linux—kernel 3.2[6]. For the case of single-thread runs, the execution time values are those achieved by simply running the application code on top of a calendar queue scheduler.

In Figure 3 we report the observed execution time values while varying the number of threads (each reported value resulting as the average over 5 different samples). The data clearly show how our mixed HW/SW approach to reversibility outperforms both the others, with a maximum gain of up to 10% vs the pure HW approach and of 30% vs the pure SW approach (achieved when running with 4 threads). Such a gain by the mixed approach is clearly related to the fact that write-intensive phases lead the pure software recoverability support to become more intrusive, because of costly generation of bigger undo code blocks, which does not pay-off compared to the reliance on pure HTM-based reversibility. On the other hand, the pure HTM-based approach does not allow the maximization

[6] The hyper-threading support offered by the processors has been excluded just to avoid cross-thread interferences—due to conflicting hyper-threads' accesses to hardware resources—which might alter the reliability of our analysis

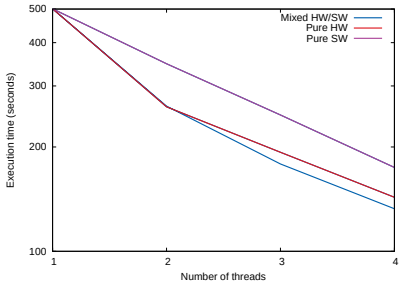

Fig. 3. Execution time - log scale on the y-axis.

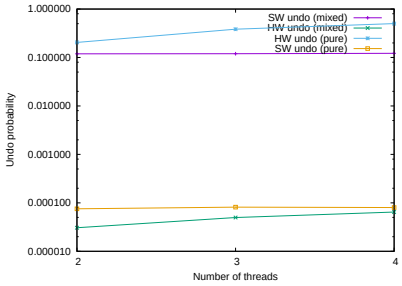

Fig. 4. Undo probability for HW and SW speculatively processed events.

of the usefulness of the carried out speculative work for larger thread counts. In fact, the slope of the execution time curve for the pure HW approach becomes slightly worse than the one of the pure SW approach when moving from 3 to 4 threads. Our mixed approach is able to get the best of the two by just avoiding excessive aborts of HTM transactions when relying on larger thread counts, also reducing the cost of undo code blocks generation thanks to a fraction of events executed with HTM support. The data reported in Figure 4 show how the pure HW approach suffers from a kind of thrashing when increasing the thread count, while the pure SW approach has minimal incidence of events undo, and that the mixed approach avoids the thrashing phenomenon just like the pure SW approach does (but has less overhead since executes a portion of the events via HTM support).

5 Conclusions

We have presented a speculative PDES engine where reversibility of causal inconsistent events is based on a mix of hardware and software facilities. The hardware part relies on HTM support offered by modern processors, particularly the Intel Haswell, while software reversibility is based on transparent instrumentation and on the dynamic generation of blocks of code able to undo memory side effects. We have shown via an experimental study with a classical benchmark how the proposed mixed approach can overcome the drawbacks of both the two baseline ones, in terms of delivered performance of by the simulation engine.

References

1. Brown, R.: Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. Communications of the ACM 31(10), 1220–1227 (1988)
2. Carothers, C.D., Perumalla, K.S., Fujimoto, R.M.: Efficient optimistic parallel simulations using reverse computation. ACM Transactions on Modeling and Computer Simulation 9(3), 224–253 (1999)

3. Cingolani, D., Pellegrini, A., Quaglia, F.: Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES. In: Proceedings of the 2015 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. PADS, ACM Press (2015)
4. Dice, D., Shavit, N.: TLRW: Return of the Read-write Lock. Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures pp. 284–293 (2010)
5. Ferscha, A.: Probabilistic Adaptive Direct Optimism Control in Time Warp. In: Proceedings of the 9th Workshop on Parallel and Distributed Simulation. pp. 120–129. IEEE Computer Society (1995)
6. Ferscha, A., Luthi, J.: Estimating Rollback Overhead for Optimism Control in Time Warp. In: Proceedings of the 28th Annual Simulation Symposium. pp. 2–12. IEEE Computer Society (apr 1995)
7. Fujimoto, R.M.: Parallel Discrete Event Simulation. In: Communications of the ACM. WSC, vol. 33, pp. 19–28. ACM Press (1989)
8. Fujimoto, R.M.: Performance of Time Warp Under Synthetic Workloads. In: Proceedings of the Multiconf. on Distributed Simulation. pp. 23–28. Society for Computer Simulation (1990)
9. Fujimoto, R.M., Tsai, J.J., Gopalakrishnan, G.: Design and Evaluation of the Rollback Chip: Special Purpose Hardware for {Time Warp}. IEEE Transactions on Computers 41(1), 68–82 (1992)
10. Jefferson, D.R.: Virtual Time. ACM Transactions on Programming Languages and System 7(3), 404–425 (1985)
11. Jha, V., Bagrodia, R.: Simultaneous Events and Lookahead in Simulation Protocols. ACM Transactions on Modeling and Computer Simulation 10(3), 241–267 (2000), http://doi.acm.org/10.1145/361026.361032
12. LaPre, J.M., Gonsiorowski, E.J., Carothers, C.D.: LORAIN: a step closer to the PDES 'holy grail'. In: Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of Advanced Discrete Simulation. pp. 3–14. PADS, ACM Press, New York, New York, USA (2014)
13. Mehl, H.: A deterministic tie-breaking scheme for sequential and distributed simulation. In: Proceedings of the Workshop on Parallel and Distributed Simulation. ACM (1992)
14. Pellegrini, A.: Hijacker: Efficient static software instrumentation with applications in high performance computing: Poster paper. In: Proceedings of the 2013 International Conference on High Performance Computing and Simulation, HPCS 2013. pp. 650–655. Helsinki, Finland (2013)
15. Pellegrini, A., Vitali, R., Quaglia, F., Pellegrini, A., Quaglia, F.: Autonomic State Management for Optimistic Simulation Platforms. IEEE Transactions on Parallel and Distributed Systems 26(6), 1560–1569 (2015)
16. Preiss, B.R., Loucks, W.M., MacIntyre, D.: Effects of the Checkpoint Interval on Time and Space in Time Warp. ACM Transactions on Modeling and Computer Simulation 4(3), 223–253 (1994)
17. Quaglia, F., Baldoni, R.: Exploiting Intra-Object Dependencies in Parallel Simulation. Inf. Process. Lett. 70(3), 119–125 (1999)
18. Quaglia, F., Santoro, A.: Non-Blocking Checkpointing for Optimistic Parallel Simulation: Description and an Implementation. IEEE Transactions on Parallel and Distributed Systems 14(6), 593–610 (2003)
19. Santini, E., Ianni, M., Pellegrini, A., Quaglia, F.: HTM Based Speculative Parallel Discrete Event Simulation of Very Fine Grain Models. In: Proceedings of the 22nd International Conference on High Performance Computing (HiPC). HiPC (2015)

20. Soliman, H.M., Elmaghraby, A.S.: An Analytical Model for Hybrid Checkpointing in Time Warp Distributed Simulation. IEEE Transactions on Parallel and Distributed Systems 9(10), 947–951 (1998)
21. West, D., Panesar, K.: Automatic Incremental State Saving. In: Proceedings of the 10th Workshop on Parallel and Distributed Simulation. pp. 78–85. PADS, IEEE Computer Society (1996)