Hardware-Transactional-Memory Based Speculative Parallel Discrete Event Simulation of Very Fine Grain Models

Emanuele Santini, Mauro Ianni, Alessandro Pellegrini, Francesco Quaglia
DIAG–Sapienza Università di Roma, Italy
Email: emalele1688@gmail.com, mauroianni@gmail.com, {pellegrini,quaglia}@dis.uniroma1.it

***Abstract*—This article presents an innovative runtime support for speculative parallel processing of discrete event simulation models on multi-core architectures, which exploits Hardware-Transactional-Memory (HTM) facilities for the purpose of state recoverability. In this proposal, the speculative updates on the state of the simulation model are executed as concurrent HTM-based transactions that are also in charge of detecting whether the update is consistent with the advancement of logical-time along model execution. Our proposal is fully transparent to the application code. Hence, our HTM-based run-time support can host conventionally developed discrete event models relying on the concept of event-handlers to be dispatched by an underlying simulation engine. Experimental data show that our proposal provides 75% to 92% of the ideal speedup on an Intel Haswell based platform (equipped with 4 physical cores and HTM support) for discrete event models with event granularity ranging between 2 and 12 microseconds. The data also show that these same models cannot be executed efficiently on top of a last generation parallel discrete event simulation platform employing software-based recoverability.**

***Keywords*-state recoverability; optimistic PDES;**

I. Introduction

In Parallel Discrete Event Simulation (PDES) [1], the simulation model is partitioned into simulation objects—historically referred to as Logical Processes (LPs)—that are allowed to be dispatched for event processing along concurrent worker-threads. This allows for exploiting hardware parallelism with the aim at speeding up model execution. The simulation object is usually implemented as a set of data structures to be updated via a callback (representing the application entry point), which is dispatched by the underlying PDES platform (see, e.g., [2]–[4]). The dispatch operation corresponds to the processing of a timestamped event at the simulation object, and causally consistent execution is typically based on forcing any simulation object to process its input events in non-decreasing timestamp order, including those produced by other objects as the result of processing activities they carried out. In fact, although recent approaches provide alternative object-interaction methods (see, e.g., [5], [6]), the cross-scheduling of events among simulation objects is the basic approach adopted in PDES to model the interactions occurring among the entities belonging to the simulated system/scenario.

In speculative PDES [7] there is no preliminary assessment of causal consistency of the events, rather they are dispatched for execution on whichever simulation object as soon as they are available. This leads to high exploitation of the intrinsic parallelism in the simulation model, since causally unrelated portions of the simulated state trajectory can be processed with no a-priory synchronization of the execution of the different simulation objects. However, if the updates occurring along a computation path are eventually detected to be inconsistent (i.e. they occurred out of timestamp order), rollback mechanisms need to be actuated so as to restore the application state to a consistent (past) snapshot from which forward computation can be resumed.

Although simple in principle, state recoverability of the simulation objects poses problems on the side of both performance and application transparency. In fact, the more efficient the recoverability support, the lower its overhead. On the other hand, application-transparent state restore typically demands more operations from an underlying recoverability layer, which further biases the tradeoff away from pure performance optimization. Literature studies have (jointly) addressed performance and transparency aspects in state recoverability of simulation objects via disparate checkpointing techniques [8], [9] that, except for a few proposals based on (either conventional or non-conventional) hardware support [10], [11], rely on software implementations of the checkpointing support. Although most of these proposals also entail overhead minimization techniques (e.g. via tuning of the parameters driving both checkpointing and–consequently–state recovery operations), for the case of very fine grain simulation models, namely models based on events that require a few microseconds of CPU-time for being processed, the overhead can still represent an impairment to performance. A way to cope with this issue is the alternative recoverability technique based on reverse computing [12], where the forward application code is coupled with a (in some cases automatically generated [13]) reverse code version that is used to undo the state updates that are eventually revealed to be inconsistent. This solution pays off especially in contexts where, beside having fine grain reverse (hence forward) events, the portion of the state trajectory to be undone (namely the so called *rollback length*) is short, which leads to a reduced number of reverse events to be processed per rollback operation.

Another aspect that plays a relevant role in case of speculative PDES of very fine grain models is the cost associated

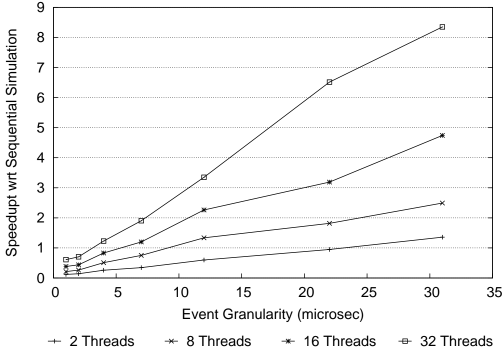

Figure 1. Speedup for PHOLD while varying event granularity and number of threads

with cross-simulation-object scheduling of events, which may become predominant. This is typically achieved via message exchange (managed at the level of the underlying PDES platform), and the classical approach to undo the notification of an event that has been scheduled as a result of the processing of another event that is then detected to be non-consistent is to send a negative copy of it (the so called *anti-message*). Beyond potentially triggering a rollback operation at the recipient (in case the original copy of the message—namely event—was already processed) anti-messages lead to doubling the communication cost per-incorrect-scheduled events. To cope with this issue, literature approaches have been proposed in order to reduce the number of message exchange operations, such as the ones based on message aggregation [14] or lazy-cancellation (lazy-antimessages) [15].

In any case, despite the existence of a bunch of literature results on optimizing speculative PDES systems, executing simulation models with very fine grain events on top of these systems in a performance-efficient manner is still a non-trivial achievement. Just to provide the reader with some empirical evidence, we report in Figure 1 the speedup achievable by running the classical PHOLD benchmark for PDES systems [16] (in a configuration with 2048 simulation objects) on top of the ROOT-Sim last generation speculative PDES platform ([1]) hosted on a 32-core off-the-shelf HP ProLiant machine, with respect to the sequential simulation of the same benchmark (same code) on a calendar-queue scheduler (still executed on the same machine). In the plot, the CPU-demand by PHOLD events is varied from a few to some tens of microseconds. The plotted curves show that speedup is unacceptable (it is a slow-down) for minimal CPU-requirements by the events, and is anyhow non-competitive (vs the employed number of threads) even when events last tens of microseconds.

In this article we cope with the issue of speculatively running (very) fine grain PDES applications efficiently on top of multi-core machines, which is achieved by exploiting the Hardware-Transactional-Memory (HTM) support that is nowadays offered by off-the-shelf processors (such as the Intel Haswell). Overall, our proposal is suited for contexts where conventional speculative PDES engines based on software recoverability (even the most advanced ones) fail to provide speedup just due to the excessively fine granularity of the simulation events (as we have shown in Figure 1).

In our proposal we speculatively execute an event as an HTM-based transaction that includes the actual buffering of any newly produced event destined to whichever simulation object (in case the transaction is successfully committed), and which entails a code-block that is used to explicitly detect whether the transaction (hence the processed event) is causally consistent, so that a commit is issued only in case consistency is verified. On the other hand, if the transaction is not guaranteed to be causally consistent, it simply issues an abort command that allows: (A) automatically undoing the updates issued on the state of the simulation object and (B) automatically discarding any new event produced as a result of incorrect event processing. Both these targets are achieved with no intervention by any software layer thanks to the fact that the HTM transactional cache keeping the state updates and the newly produced events is simply squashed upon the abort of the transaction. A secondary effect by our approach is that we allow intra-simulation-object concurrency in the speculative processing scheme, as opposed to traditional PDES engines where a single event at a time can be CPU-scheduled for a specific simulation object. Specifically, in our approach two worker-threads operating within the PDES environment are allowed to concurrently execute two different events targeting the same simulation object (just depending on how the overall set of events destined to the different simulation objects is clustered along the simulation time axis). If the two events are actually independent (given that their read/write sets are disjoint, which means that they touch different portions of the simulation object state) then they are both committable in our execution model. This leads our system to implement the so called weak-causality model [17], just in the form of parallelization of the execution of events within the same simulation object. As a last note, our approach is application transparent given that the application layer can still be designed as a set of event-handlers touching application specific data structures, as it commonly occurs in reference speculative PDES environments (see [2]–[4]) not relying on HTM facilities and sequential simulators as well.

We also report the results of an experimental study based on running our system, and specific test-bed applications, on top of a machine equipped with 2 quad-core (hyper-thread) Intel(R) Xeon(R) 3.5 GHz processors (with HTM support) and 24 GB RAM, running Linux Ubuntu 12.04.2 LTS, kernel version 3.5.0-23-generic. By the data, our HTM-based simulation engine allows achieving 75% to 92% of the ideal speedup and a performance gain of up to 10x vs

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

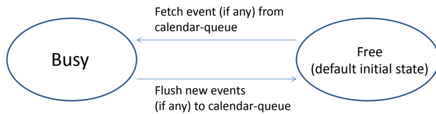

Figure 2. State diagram for the generic worker-thread WT_i

the last generation ROOT-Sim speculative PDES platform relying on software-based recoverability support.

The remainder of this article is structured as follows. Our HTM-based speculative PDES engine is presented in Section II. Experimental data are provided in Section III. Related work is discussed in Section IV.

II. The HTM-based Speculative PDES Engine

We consider a scenario where all the events that have been scheduled (including the simulation startup events), destined to whichever simulation object, are kept within a unique pool implemented as a classical calendar-queue [18]. The events kept within the pool are extracted for concurrent processing by multiple worker-threads.

The sorting of the elements into the calendar-queue is based on event timestamps, but event-records also entail information determining what simulation object is the target of an event, and the actual event type/payload, as typical of PDES environments.

All the events kept in the calendar-queue are *schedule-committed*, with the meaning that they will never be retracted (e.g. via negative copies). In fact, in our approach events that were scheduled during an HTM-supported event processing phase are flushed to the calendar-queue only if the transaction associated with the processing of the event successfully commits. This, in its turn, implies that the event is no longer rollbackable, therefore the output it produced will never need to be undone.

We target the scenario where the maximum number of worker-threads employed in the speculative PDES engine is upper bounded by the number of available CPU-cores, say N. This is a classical configuration avoiding interference by deschedule/reschedule operations in parallel applications [19]–[21] at least for cases where the platform is temporarily dedicated to a specific application.

The calendar-queue data structure is coupled with an array of N entries, which we name `processing[]`. The i-th entry of this array is used to keep data, say a timestamp value, related to the status of the i-th worker-thread operating within the speculative PDES platform, which we denote as WT_i. This array is initialized at simulation startup in such a way to keep in all the entries the special value ∞. On the other hand, the calendar-queue is initialized in such a way to keep the initial events that, depending on the simulation model configuration, will fire any initial state transition in the model execution.

Algorithm 1 Fetch operation - worker-thread WT_i

1: **procedure** FETCH [ATOMIC]RETURNS: event
2: $e \leftarrow$ GETMINIMUMTIMESTAMPEVENTFROMCALENDARQUEUE
3: **if** $e = NULL$ **then**
4: `processing[i]` $\leftarrow \infty$
5: **else**
6: `processing[i]` $\leftarrow T(e)$
7: **end if**
8: return e
9: **end procedure**

Each worker-thread WT_i lives in the state diagram shown in Figure 2. It starts executing within the *free* state, with the meaning that it has no pending (to be executed) simulation event yet assigned to it. Hence, WT_i initially has no simulation object to take care of (for event processing). WT_i leaves the *free* state and enters the *busy* one upon performing a FETCH operation that leads to the extraction of some event to be processed from the calendar-queue. This operation executes the actions described in Algorithm 1. Specifically, it atomically extracts the event e with minimum timestamp that is currently registered into the calendar-queue (if any), and records the extracted timestamp value into the entry `processing[i]` associated with WT_i. Atomicity avoids that two different worker-threads take care of processing the same pending event. Also, if two worker-threads, say WT_i and WT_j, execute the FETCH operation concurrently, and the two operations are serialized in such a way that the two threads extract from the calendar-queue, respectively, the event e and then the event e', it is guaranteed that $T(e) < T(e')$ ([2]). Hence it is also guaranteed that `processing[i]<processing[j]`, given that the two array entries are updated according to the established serialization order. We also note that the FETCH operation can be executed in constant-time average performance, since the calendar-queue guarantees $O(1)$ average-performance. Hence, the usage of a conventional spin-lock to achieve atomicity of the FETCH operation should not represent a scalability impairment, at least at the CPU-core count that currently characterizes processors offering HTM support ([3]).

If no event is extracted from the calendar-queue by WT_i while executing the FETCH procedure (i.e. the calendar-queue is found to be empty), the entry `processing[i]` is set to the default initial value ∞. In the main loop of simulation processing, this scenario will simply lead to retry the FETCH operation, leaving the worker-thread in the free state, given that no job to perform has been assigned to it.

When WT_i is allowed to commit the event it is currently handling (while in the *busy* state), any speculative operation (e.g. memory update) associated with event processing,

[2]Here we implicitly assume that no simultaneous events will ever exist. However, the case of simultaneous events, where $T(e)$ may be equal to $T(e')$, will be explicitly dealt with later in the article.

[3]For scaled-up CPU-core counts, as we may expect it will be the case for next-generation HTM-equipped machines, we can envisage the reliance on wait-free algorithms rather than lock-based ones [22], [23].

Algorithm 2 Flush operation - worker-thread WT_i

```
1: procedure FLUSH [ATOMIC](event_set E)
2:     ∀e ∈ E: InsertInCalendarQueue(e)
3: end procedure
```

which it kept at the level of the HTM cache, is allowed to be flushed to memory for making it visible. In our organization, the HTM-based transaction that implements event processing writes newly scheduled events (possibly produced by the current event execution) into a thread-private buffer, such that the buffer content is made visible only upon committing the HTM-based transaction. Hence, right after committing the event processing phase, these events can be flushed (outside of the transactional code-block) from the thread-private buffer to the calendar-queue. Overall, the activities carried out by any worker-thread WT_i right after committing some event, which we globally refer to as FLUSH procedure, are the ones depicted in Algorithm 2. This procedure is still an atomic action (hence it can rely on, e.g., the same spin-lock used for managing the FETCH operation) and simply inserts all the newly produced events (if any) into the calendar-queue. Upon executing this procedure, the worker-thread switches back to the *free* state.

The flushed events will be eventually extracted from the calendar-queue by any thread that will (re)transit into the *free* state, via the execution of FETCH operations. Clearly, a new event e' that is flushed to the calendar-queue might be associated with a timestamp $T(e')$ such that $T(e') < T(e)$, where e is an event previously fetched by some worker-thread WT_j (namely, the one with minimum timestamp across those stored in the calendar-queue at the time of the FETCH operation). In this case, the newly scheduled event e' stands in the past of e, which plays a role in the determination of event safety (causal consistency) within the speculative processing scheme.

To cope with the consistency issue, the values registered in the array `processing[]` are used in our approach to define the order according to which the events currently handled by the worker-threads (while being in the *busy* state) need to be committed. Specifically, they establish the order according to which the HTM-based transactions implementing the processing of the events need to be committed. Hence, we use the array entries in a manner similar (at least in spirit) to what is done by Lamport's bakery algorithm [24]. Overall, the condition that tells whether a worker-thread WT_i can safely commit the event it is handling is expressed as:

$$\forall j \neq i : \texttt{processing}[\texttt{i}] < \texttt{processing}[\texttt{j}]$$

This condition indicates that the (possibly speculatively) executed event is associated with the current lower bound timestamp across all the not yet processed/committed events in the system. Hence the timestamp of this event represents the commit horizon, thus the event can be safely executed or

Algorithm 3 Safety-check - worker-thread WT_i

```
1: procedure SAFE RETURNS: BOOLEAN
2:     T̂ ← MIN_{∀j≠i}(processing[j])
3:     if (processing[i] < T̂) then
4:         return TRUE
5:     else
6:         return FALSE
7:     end if
8: end procedure
```

(in case of already carried out speculative execution) safely committed. The pseudo-code implementing the safety-check is provided in Algorithm 3. As the reader may observe, the SAFE procedure does not require to be executed atomically, given that when any worker-thread WT_j executes the FLUSH procedure, it leaves `processing[j]` untouched.

From the above arguments, the actual execution loop of any worker-thread WT_i is the one reported in Algorithm 4, where the safety-check is initially carried out before any choice is taken in relation to the way (speculative or not) the event e currently assigned to WT_i needs to be processed. If the safety-check at line 7 is verified, then the event can be executed with no need to startup the HTM-based transaction for recoverability purposes, since any side effect the event would give rise to is safe. In other words, the timestamp of the event is already known (prior to the actual processing of the event) to correspond to the commit horizon of the speculative run. On the other hand, if the safety-check at line 7 is not verified, WT_i processes the event speculatively within an HTM-based transaction (hence in a recoverable manner, given that the transaction can be aborted, if needed). After the processing phase, the safety-check is re-executed and if the event has become committable (namely, it now lies on the—hopefully—advanced commit horizon) then the actual commit takes place, with installation of the event side effects that become visible. In the negative case, the whole process of safety-assessment and safe vs speculative execution is retried (resuming from line 7). In other words, the structure of Algorithm 4 leads any worker-thread WT_i to be able to execute its currently assigned event in safe mode or in speculative mode. In the latter case, we have chances that, at the end of the speculative processing phase, the event has become committable, thanks to the absence of non-committed events still standing in its past.

A. Optimizations

Handling Simultaneous Events: The safety condition discussed above is based on having some worker-thread WT_i in the *busy* state that is in charge of processing (or has speculatively processed) the event with the current absolute minimum timestamp within the whole system. This condition might never be verified with simultaneous events, namely events marked with the same identical timestamp. If simultaneity of events were admitted in the simulation model, then Algorithm 4 would give rise to live-lock.

Algorithm 4 Main loop

```
1: procedure MAINLOOP
2:     while ¬endSimulation do
3:         e ← FETCH( )
4:         if e = NULL then
5:             retry from line 3
6:         end if
7:         if SAFE then
8:             event_set New_events ← PROCESSEVENT(e)
9:             FLUSH(New_events)
10:        else
11:            BEGINTRANSACTION( )
12:            event_set New_events ← PROCESSEVENT(e)
13:            if SAFE then
14:                COMMITTRANSACTION( )
15:                FLUSH(New_events)
16:            else
17:                ABORTTRANSACTION( )
18:                retry from line 7
19:            end if
20:        end if
21:    end while
22: end procedure
```

As for this aspect, we note that guaranteeing progress in speculative PDES systems in the presence of simultaneous events is a well understood problem that has been extensively studied in literature [25], which is not specifically bound to our proposal. Hence different literature solutions for tie-breaking simultaneous events (see, e.g., [26]) can be exploited for integration with our HTM-based speculative PDES system. A baseline approach could consist in comparing both timestamp values and worker-thread identifiers, according to the philosophy underlying Lamport's bakery algorithm [24]. Therefore, a variant for safety-assessment (of the event currently handled by any worker-thread WT_i) in the presence of simultaneous events, which we already implemented in our engine, is based on the following predicate:

$$\forall j \neq i : \texttt{processing}[\texttt{i}] \leq \texttt{processing}[\texttt{j}] \texttt{ AND } i < j$$

This variant does not consider (possible) causal relations across simultaneous events, since the tie-break is exclusively based on worker-threads' identifiers. However, the achievement of liveness with simultaneous events, while jointly guaranteeing causality across them, could be reached in our scheme by relying on causal-timestamps (see, e.g., [27]). Hence `processing[]` could be simply setup to keep causal-timestamps rather than classical ones if a scenario with causal simultaneous events would need to be dealt with.

Non-zero Lookahead: From the PDES literature it is well known that the simulation model *lookahead* can play a role on the efficiency of synchronization ([4]). Although it plays a major role for conservative PDES [1], it can play such a role also in speculative PDES systems. However, the traditional way the lookahead is used is to determine (a-priori for conservative PDES vs a-posteriori for speculative PDES) the safety of the events that are executed by a simulation object when assuming that the object is a sequential entity. In our approach, objects are no longer sequential entities, given that two different worker-threads can contemporaneously reside in the *busy* state by having fetched two events destined to the same simulation object. Let us indicate with $T(e)$ and $T(e')$ the timestamps of these two events and assume, with no loss of generality, that $T(e) < T(e')$. Suppose, still with no loss of generality, that the simulation model has lookahead L, such that $T(e) + L > T(e')$. In such a scenario, we cannot assert that the event e' is safe (which might lead it to commit before e is committed), given that it may need to access the simulation object snapshot that has been produced by e. Overall, event safety calculation on the basis of the lookahead can be still adopted for scenarios where the event in the past, say e in our example, is associated with a simulation object different from the one associated with the event in the future, say e' in the same example. In such a scenario, the condition $T(e) + L > T(e')$ leads to the fact the any new event produced by the source simulation object, the one processing e, will not be causally related to e'. Hence, the overall HTM-based speculative PDES engine organization can be modified by including an additional array `destination[]`, with N entries, such that, in case WT_i is in the *busy* state, `destination[i]` keeps the identifier of the simulation object that is the target of the event to be processed by WT_i. This way we can distinguish a priori whether different events that are concurrently handled by two worker-threads operate on disjoint portions of the simulation model. By exploiting this new array, and the lookahead value L (if any), the safety of the event to be processed by WT_i can be assessed according to the following condition:

$$\begin{aligned} &\forall j \neq i \texttt{ such that destination}[\texttt{j}] \neq \texttt{destination}[\texttt{i}] : \\ &\qquad \texttt{processing}[\texttt{i}] < \texttt{processing}[\texttt{j}] + L \\ &\quad \texttt{AND} \\ &\forall j \neq i \texttt{ such that destination}[\texttt{j}] = \texttt{destination}[\texttt{i}] : \\ &\qquad \texttt{processing}[\texttt{i}] < \texttt{processing}[\texttt{j}] \end{aligned}$$

The above logic can increase concurrency by allowing the (speculatively) processed events to be committed in non-strictly increasing values of their timestamps, as instead it needs to occur when relying on the condition adopted by the SAFE procedure in Algorithm 3. Further, the above predicate can be still integrated with the aforementioned logic for managing simultaneous events based on worker-thread identifiers. As a last note, such a predicate is fully independent of the actual interaction graph across the simulation objects. This is a choice aligned with the classical objectives of speculative PDES synchronization, which does not require a-priori knowledge of the (potential) interactions

[4]If a discrete event model has *lookahead* value L, then it is guaranteed that any event with timestamp $T(e)$ will not give rise to any other event with timestamp less than $T(e) + L$. Hence e will not give rise to causality dependencies up to time $T(e) + L$. Overall, the lookahead is the ability to predict that nothing will occur in logical time up to a point that is a function of the current time.

across the concurrent simulation objects.

Throttling: Another optimization we discuss is related to line 13 of Algorithm 4. In this line of the pseudo-code the SAFE procedure is called while being in HTM-based transactional context. It is not useful to call this procedure multiple times (e.g. according to a polling approach for safety assessment) within the same transaction while handling event e. This is because any update occurring onto the array checked by SAFE leads to the abort of the HTM-based transaction associated with the processing of e in case the safety-check was previously invoked within the same transaction and the array was updated via FETCH operations by concurrent worker-threads. In order to increase the likelihood that, in case e has become a safe event along its speculative processing interval, we can actually detect its safety upon calling the SAFE procedure at line 13 of Algorithm 4, we have devised a throttling approach (which we recall is a classical technique for reducing the likelihood of rollbacks in speculative PDES, see, e.g. [28]).

In particular, we modified the procedure SAFE to return to the invoking worker-thread WT_i currently handling event e, the number of entries of the `processing[]` array which keep timestamps (possibly augmented with the lookahead value) that are lower than the timestamp kept by `processing[i]`. We denote with K this number, which corresponds to the minimum number of events that need to be committed before a speculative run of the event e bound to WT_i can be committed ([5]). Clearly, when calling SAFE at line 7 of Algorithm 4, K is zero if the event bound to WT_i is safe, while K is different from zero in case the event is not detected to be safe. However, in case the event e is not yet safe, K provides an indication of the minimum number of events from which e may causally depend.

In our throttling scheme, we insert a delay (a CPU-busy loop, since operating system sleep cannot be used, given that it would lead to aborting HTM-based transactions because of a mode-change along thread execution) which is computed as $\delta \times K \times \alpha$, where δ corresponds to the average event granularity for the executed simulation model and α is a parameter with value falling in the interval [0,1] which is determined according to a classical hill-climbing approach, similar to the one adopted in [29] for tuning the checkpoint interval in PDES platforms to the value that optimizes performance. In our hill-climbing approach the metric used to dynamically set α is the number of committed events per wall-clock-time unit. In other words, we regulate throttling in such a way to increase the likelihood of performing useful work while the worker-threads reside in the *busy* state. We also included an ϵ-greedy scheme to avoid stalling in local maxima.

[5]It is the minimum, not the exact value, because K does not account for events with timestamps lower than `processing[i]`, if any, which might have been already inserted into the calendar-queue, but that might have not been fetched for processing.

	Speculative engine type	
	classical	HTM-based
suitability for very fine-grain events	no (or limited)	yes
unbounded chain of speculative events	yes	no
intra-simulation-object parallelism	no	yes
suitability for very large scale platforms	generally yes	no

Table I
SUMMARY OF HTM-BASED VS CLASSICAL SPECULATIVE PDES.

B. HTM-based vs Classical Speculative PDES: a Summary

We provide in Table I a brief summary of the main differences (as evaluated by relying on four reference indices) between our HTM-based approach and classical speculative PDES. The latter allows for (ideally) unbounded chains of speculatively processed events along the execution path of each individual simulation object (it may only depend on memory limits for keeping speculative and recoverability data), while our HTM-based approach allows for up to N (number of CPU-cores) speculative events to stand out, given that recoverability relies on the hardware transactional cache. On the other hand, our approach allows for intra-simulation-object concurrency while classical speculative PDES does not allow for it. In fact, our engine automatically resolves data conflicts arising while processing in parallel events destined to the same simulation object (this leads to squash/retry of the corresponding HTM-based transactions in case some conflict materializes). As for usefulness when parallelizing very fine grain models, the HTM-based approach fully fits it, while the traditional approach may provide limited usefulness. On the other hand, the classical approach has been already shown to scale to very large computing platforms (see, e.g., [30]), while our solution is intrinsically targeted at limited scale machines (e.g. because of the atomicity requested in manipulating the shared calendar-queue across the worker-threads, or the shared array of meta-data). However, at current date, machines with HTM support show relatively limited number of cores. Hence our approach looks suited for current HTM platforms. Also, for very fine grain models, significant reduction of the completion time can be achieved even with limited (but well exploited) amounts of CPU-cores, which is one target we achieve, as shown by the experimental data we provide in the following section.

III. EXPERIMENTAL RESULTS

We initially tested our HTM-based speculative PDES system by relying on the well known PHOLD benchmark [16]. We included 2048 simulation objects 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 different objects. 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. For this benchmark configuration, we varied the CPU-demand for processing the events in the interval between 2 and 12 microseconds, which has been done by appropriately setting the classical busy-loop characterizing PHOLD event processing steps. Hence we studied the system behavior when moving from very fine to fine grain event configurations. We have run this benchmark by varying the number of employed threads from 1 to the maximum number of physical CPU-cores, say 4, in the underlying HTM-equipped machine (which entails two 4-core Intel Haswell 3.5 GHz processors with hyper-threading support and has 24 GB RAM)([6]). For the case of single-thread runs, the execution time values are those achieved by simply running the application code on top of the calendar-queue scheduler, while for all the other settings of the number of threads we relied on the HTM-based parallel implementation we presented ([7]). We have also run the same identical model on top of the last generation ROOT-Sim speculative PDES engine, which offers a pure software-based support for recoverability. In Figure 3 we report the observed speedup values vs the single-thread execution time (each reported value resulting as the average over 5 different samples). The data clearly show how our HTM-based proposal definitely outperforms the traditional style PDES engine. Particularly, our solution allows achieving speedup that ranges between 1.5 and 3.6, with the highest values achieved when the granularity of the events increases towards 12 microseconds. Also, with 4 threads it provides speedup above 3 as soon as the event granularity is of at least 4 microseconds. Instead, the traditional style PDES engine only provides slow-down, which again confirms the unsuitability of the classical software-based recoverability support for speculative execution of models with very reduced event granularity.

In Figure 4 we report an additional set of data showing how the HTM-support for event speculation influences the execution dynamics. In particular, we show the probability that some event gets eventually committed after having been executed speculatively within an HTM-based transaction. The data show an interesting trend where for lower levels of parallelism (say 2 threads) the HTM support does not influence speculation (and its performance effects) significantly. In fact, the likelihood for a committed event to have been executed within an HTM-based transaction is very low, indicating how the most important contribution to parallelism in the execution is provided by the exploitation of the lookahead, which allows for processing events in a safe mode outside any transaction. On the other hand, when increasing the level of parallelism, the HTM-based support starts to play a relevant role, given that the probability for a committed event to have been processed speculatively within some HTM-based transaction increases up to almost 0.2. Also, the increase of the usefulness of HTM-based speculative processing when moving from 2 to 4 threads indicates a potential for scalability of our approach to HTM-equipped machines with larger numbers of physical cores.

Another set of experiments has been carried out by relying on a multi-robot (multi-agent) exploration and mapping simulation model, developed on the basis of the results in [31]. Specifically, in this model a group of robots is set out into an unknown space, with the goal of fully exploring it, while acquiring data from sensors (e.g., cameras, lasers, ...) which are used to map the environment. Whenever a robot has to make a decision about which direction should be taken to carry on the exploration, it is done by relying on *pheromones count*. Specifically, each subregion is assigned a counter which is incremented whenever any robot visits it so as to notify other ones of its transit. To decide what direction to take, robots adopt a greedy approach, so that when a robot is in a particular subregion, it targets the neighbor with the minimum trail count. A random choice takes place if multiple subregions have the same (minimum) trail count. In our implementation of this model, each simulation object models a squared subregion of an overall region to be explored. We considered 2025 subregions, that are explored in parallel by 16 robots, modeling the scenario of a relatively reduced number of high-qualified agents in charge of the exploration. Each robot has a mean residence time within a subregion of 5 min, and for physical constraints it cannot pass though a subregion in less than 30 seconds, a value that determines the lookahead of the model (since a new arrival event in any subregion cannot occur before 30 seconds have elapsed since the arrival in the currently visited subregion). This simulation is aimed at determining the coverage time, depending on the choices that are performed while determining what new subregion to enter. This model is still very fine grain given that mobility events only entail determining what direction to choose. The results for this application are presented in Figure 5. By the data we can see how the HTM-based solution is able to deliver maximum speedup of the order of (slightly less than) 3.5, just when using 4 threads. Also, it provides super linear speedup when employing 3 threads. Further, the traditional style PDES engine did not provide any reasonable speedup also for this case study, rather a slow-down. This additionally supports the relevance of our HTM-based proposal.

Finally, in Figure 5 we also report a speedup curve achieved when using a lookahead for event safety detection

[6]We did not use hyper-threading (hence the whole available set of 8 cores) in order to avoid interference by different threads on transactional-cache portions that are shared across hyper-threaded cores, which would likely lead HTM-based speculative processing of events to abort due to phenomena that are not directly imputable to our speculative processing support. Data gathered during our experimentation confirm this expectation.

[7]This is available as open source at https://github.com/HPDCS/htmPDES.

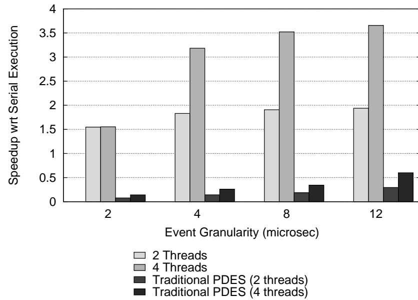

Figure 3. Speedup values for PHOLD

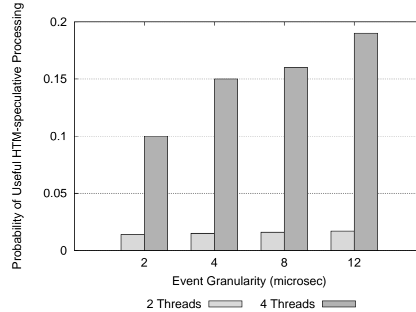

Figure 4. Usefulness of HTM-based speculation

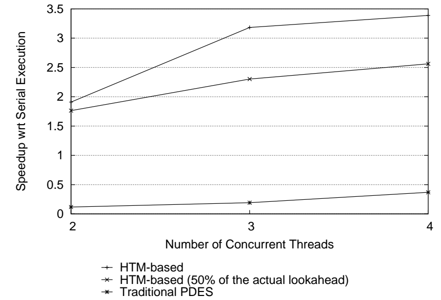

Figure 5. Speedup values for the multi-robot model

that has been set to 50% of the actual lookahead of the simulation model. By the data we see that this configuration still provides good speedup, which for the case of 4 threads is slightly less than the 80% of the value observed when employing the real lookahead of the application. This is an additional indication of the usefulness of HTM-based speculation, especially for larger numbers of threads (as we have already noted for the PHOLD benchmark). Finally, these data show how the HTM-based approach can provide resilience to performance failures in scenarios where the lookahead managed at the level of the simulation engine is an under-estimation of the real one. This might help setting up the engine in scenarios where the determination of the precise lookahead of the application can be a time consuming job (e.g. when not relying on automatic approaches to lookahead extraction [32]).

IV. Related Work

As hinted, most of the state recoverability proposals in speculative PDES are based on software approaches. The ones relying on checkpointing tend to reduce the overhead of the recoverability support (and of the actual recovery operations executed in case of rollback) by either exploiting incremental or sparse checkpointing techniques [8], [33], [34], or a combination of the two [9], [35]. Compared to these schemes, our proposal is orthogonal since we guarantee recoverability by relying on a hardware support, rather than a software one.

The recent proposal in [36] discusses the possibility to speculatively execute discrete event simulation applications in parallel on top of shared-memory multi-core systems by exploiting the TM paradigm as the means for state recoverability. However, this work is still bound to software based recoverability, and does not attempt to exploit the innovative HTM-support provided by mainstream processors, as instead we do in our proposal.

Other checkpointing schemes oriented to speculative PDES have been based on hardware level facilities. The proposal in [10] presents the rollback chip, which is a special purpose device used to keep the live state of the simulation object and entails the capability to perform the restore of past values. Compared to this scheme, our proposal does not require specialized hardware, rather it is based on off-the-shelf general purpose HTM facilities. An alternative way of using hardware support for state recoverability in speculative PDES has been provided in [11], where the actual checkpoint operation (data copy from live state to the checkpoint buffer) is realized in non-blocking manner via software managed DMA engines. In our proposal we retain the same non-blocking advantage, in fact the before image of the simulation object state is implicitly guaranteed to be available even when the simulation object is CPU-dispatched, thanks to the underlying hardware transactional cache used to host the after image associated with event processing. On the other hand, the restore to the before image in our proposal does not require software intervention, given that it only entails aborting the transaction associated with the incorrectly processed event. This is not the case for the approach in [11], which is based on software modules used to access the log and copy back the snapshot to be restored into the live state region.

Hardware supported synchronization in PDES has been also studied in [37], [38]. Both these proposals exploit hardware facilities to determine the commit (or committable) horizon of a parallel PDES run, hence to assess the safety of processed events (or of those to be still processed). These solutions are orthogonal to our one, given that they do not target hardware-based recoverability of the simulation model state trajectory. Also, the proposal in [37] stands as a theoretical design, given that the hardware component implementing the reduction that calculates the commit horizon has not been physically realized, rather it has only been evaluated via simulation. Instead, our engine is based on real off-the-shelf hardware facilities.

The proposal in [39] exploits HTM as the means for the atomic management of the event pool in multi-thread PDES platforms (by encapsulating concurrent accesses to the pool within HTM-based transactions). Rather, we exploit HTM for state recoverability.

As hinted, approaches explicitly tailored to fine grain speculative PDES include those oriented to reduce the communication overhead, thus ultimately improving the

computation to communication ratio. These include schemes like (i) lazy-cancellation [15], where an anti-message is sent out only after the assurance that the corresponding message to be canceled would never been recreated, (ii) message aggregation [14], where messages are sent after batching them so as to amortize the send-setup cost, (iii) zero-copy message passing [40], where the number of data copies along the path from source to destination is reduced to a minimum, and (iv) risk free synchronization [41], where produced messages (events) are sent out only after having determined the consistency of the corresponding source event. Compared to these approaches, we implicitly pursue similar objectives given that in our solution only committed output events (those produced by a committed transaction, namely a committed event execution) are actually flushed to memory. Hence we do not allow any non-committed output to live out of the hardware transactional cache, and we do not require sending anti-messages, given that only the output by committed events is reflected into memory.

Finally, compared to the reverse computing approach [12], which can be considered as a means to reduce the state recoverability cost in case of fine grain events (where software based checkpointing would induce excessive CPU-time/memory overhead), our approach provides a different tradeoff given that our HTM-based speculative PDES engine guarantees state consistency with no intervention by the software. In fact we do not rely on reverse events, rather on squashing the hardware transactional cache in the underlying HTM system, which leads the latency of the state restore operation to be independent of the length of rollback (as instead it does not occur in reverse computing schemes). On the other hand, our proposal allows for a speculative trajectory that has a number of speculative (uncommitted) steps bounded by the number of CPU-cores (namely the number of HTM caches available in the system), while reverse computing can be employed in contexts where the speculative chain of processed events does not undergo any specific constraint. In other words, in our approach optimism is limited by the available hardware resources, in terms of HTM caches, which is not the case for reverse computing.

V. Conclusions

In this article we have explored the idea of relying on Hardware-Transactional-Memory (HTM) to improve the execution speed of very fine grain parallel discrete event simulation applications. This is done by exploiting hardware supported in-memory transactions to implement the speculative execution of simulation events. If they are not ensured to be causally consistent at the time of committing the transaction, the rollback operation can be executed at reduced cost by simply squashing the transactional hardware cache. Experimental results show that this approach pays off in reducing the relative overhead of the classic software based recoverability support (e.g. software implemented checkpointing) for speculative parallel discrete event simulation. As a result, our scheme allowed to achieve speedup in the parallel execution of discrete event models with event granularity of the order of a few microseconds, a configuration typically non-effectively addressable via classical speculative parallel discrete event simulation engines relying on software based recoverability.

References

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

[2] D. E. Martin, T. J. McBrayer, and P. A. Wilsey, “WARPED: A Time Warp simulation kernel for analysis and application development,” in *Proceedings of the 29th Hawaii International Conference on System Sciences - Volume 1: Software Technology and Architecture*, 1996, p. 383.

[3] C. D. Carothers, D. W. Bauer, and S. Pearce, “ROSS: a high performance modular Time Warp system,” in *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, 2000, pp. 53–60.

[4] A. Pellegrini, R. Vitali, and F. Quaglia, “The ROme OpTimistic Simulator: Core internals and programming model,” in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, 2011, pp. 96–98.

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

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

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

[8] R. Rönngren, M. Liljenstam, R. Ayani, and J. Montagnat, “Transparent incremental state saving in Time Warp parallel discrete event simulation,” in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, 1996, pp. 70–77.

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

[10] R. M. Fujimoto, J. Tsai, and G. Gopalakrishnan, “Design and evaluation of the rollback chip: Special purpose hardware for Time Warp,” *IEEE Transactions on Computers*, vol. 41, no. 1, pp. 68–82, 1992.

[11] F. Quaglia and A. Santoro, “Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 6, pp. 593–610, 2003.

[12] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto, “Efficient optimistic parallel simulations using reverse computation,” *ACM Transactions on Modeling and Computer Simulation*, vol. 9, no. 3, pp. 224–253, 1999.

[13] J. M. LaPre, E. Gonsiorowski, and C. D. Carothers, "LORAIN: a step closer to the PDES 'holy grail'," in *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 2014, pp. 3–14.

[14] M. Chetlur, N. Abu-Ghazaleh, R. Radhakrishnan, and P. A. Wilsey, "Optimizing communication in Time-Warp simulators," in *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, 1998, pp. 64–71.

[15] A. Gafni, "Space management and cancellation mechanisms for Time Warp," *Tech. Rep. TR-85-341, University of Southern California, Los Angeles (Ca,USA)*, 1985.

[16] R. M. Fujimoto, "Performance of Time Warp under synthetic workloads," in *Proceedings of the Multiconf. on Distributed Simulation*, 1990, pp. 23–28.

[17] F. Quaglia and R. Baldoni, "Exploiting intra-object dependencies in parallel simulation," *Information Processing Letters*, vol. 70, no. 3, pp. 119–125, 1999.

[18] R. Brown, "Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem," *Communications of the ACM*, vol. 31, pp. 1220–1227, 1988.

[19] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2008, pp. 19:1–19:12.

[20] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller, "Scheduling support for transactional memory contention management," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010, pp. 79–90.

[21] S. Seelam, L. L. Fong, A. N. Tantawi, J. Lewars, J. Divirgilio, and K. Gildea, "Extreme scale computing: Modeling the impact of system noise in multicore clustered systems," in *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*, 2010, pp. 1–12.

[22] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming*, 2011, pp. 223–234.

[23] M. P. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 124–149, 1991.

[24] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Commun. ACM*, vol. 17, no. 8, pp. 453–455, 1974.

[25] V. Jha and R. Bagrodia, "Simultaneous events and lookahead in simulation protocols," *ACM Transactions on Modeling and Computer Simulation*, vol. 10, no. 3, pp. 241–267, 2000.

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

[27] R. Fujimoto, "Exploiting temporal uncertainty in parallel and distributed simulations," in *Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation*, 1999, pp. 46–53.

[28] S. Srinivasan and P. Reynolds, Jr., "Elastic time," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 2, pp. 103–139, 1998.

[29] A. C. Palaniswamy and P. A. Wilsey, "Adaptive checkpoint intervals in an optimistically synchronised parallel digital system simulator," in *Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration*, 1993, pp. 353–362.

[30] P. D. Barnes, Jr., C. D. Carothers, D. R. Jefferson, and J. M. LaPre, "Warp speed: executing time warp on 1, 966, 080 cores," in *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 2013, pp. 327–336.

[31] S. Koenig and Y. Liu, "Terrain coverage with ant robots: a simulation study," in *Agents*, 2001, pp. 600–607.

[32] E. Deelman, R. Bagrodia, R. Sakellariou, and V. S. Adve, "Improving lookahead in parallel discrete event simulations of large-scale applications using compiler analysis," in *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, 2001, pp. 5–13.

[33] F. Quaglia, "A cost model for selecting checkpoint positions in Time Warp parallel simulation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 4, pp. 346–362, 2001.

[34] B. R. Preiss, W. M. Loucks, and D. MacIntyre, "Effects of the checkpoint interval on time and space in Time Warp," *ACM Transactions on Modeling and Computer Simulation*, vol. 4, no. 3, pp. 223–253, 1994.

[35] H. Soliman and A. Elmaghraby, "An analytical model for hybrid checkpointing in Time Warp distributed simulation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 947–951, 1998.

[36] O. Dalle, "Using TM for high-performance discrete-event simulation on multi-core architectures," *Presentation at the EuroTM Workshop on Transactional Memory*, Apr. 2013.

[37] E. W. Lynch and G. F. Riley, "Hardware supported time synchronization in multi-core architectures," in *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation*, 2009, pp. 88–94.

[38] S. Srinivasan, M. J. Lyell, P. F. R. Jr., and J. Wehrwein, "Implementation of reductions in support of PDES on a network of workstations," in *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, 1998, pp. 116–123.

[39] J. Hay and P. Wilsey, "Experiments with hardware-based transactional memory in parallel simulation," in *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 2015, pp. 75–86.

[40] B. P. Swenson and G. F. Riley, "A new approach to zero-copy message passing with reversible memory allocation in multi-core architectures." in *Proceeedings of the 26th Workshop on Principles of Advanced and Distributed Simulation*, 2012, pp. 44–52.

[41] S. Bellenot, "Performance of a riskfree Time Warp operating system," in *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, 1993, pp. 155–158.