Hybrid Speculative Synchronisation for Parallel Discrete Event Simulation

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

Philipp Andelfinger
philipp.andelfinger@uni-rostock.de
University of Rostock
Rostock, Germany

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it
University of Rome Tor Vergata
Rome, Italy

ABSTRACT

Parallel discrete-event simulation (PDES) is a well-established family of methods to accelerate discrete-event simulations. However, the available algorithms vary substantially in the performance achievable for different models, largely preventing generic solutions applicable by modellers without expert knowledge. For instance, in Time Warp, the processing elements execute events asynchronously and speculatively with high aggressiveness, leading to frequent and costly rollbacks if misspeculations occur often. In contrast, synchronous approaches such as the new Window Racer algorithm exhibit a more cautious form of speculation. In the present paper, we combine these two fundamentally different algorithms within a single runtime environment, allowing for a choice of the best algorithm for different model segments. We describe the architecture and the algorithmic considerations to support the efficient coexistence and interaction of the algorithms without violating the correctness of the simulation. Our experiments using a synthetic benchmark and an epidemics model show that the hybrid algorithm is less sensitive to its configuration and can deliver substantially higher performance in models with varying degrees of coupling among entities compared to each algorithm on its own.

CCS CONCEPTS

• **Computing methodologies** → **Discrete-event simulation**; **Simulation environments**; *Massively parallel and high-performance simulations*; *Simulation theory*; • **Theory of computation** → Parallel computing models; Distributed computing models.

KEYWORDS

Time Warp, Window Racer, PDES, Synchronization

1 INTRODUCTION

Parallel Discrete Event Simulation (PDES) is an umbrella term encompassing an ensemble of techniques and methodologies to efficiently support large-scale simulations in parallel or distributed environments. PDES has seen an explosion of proposals to improve simulation performance over the past 40 years [15] and a wide range of application scenarios that exploit it to reduce model time-to-solution.

A broad spectrum of different algorithms and protocols can support the execution of PDES simulations [41]. This spectrum is characterised by different attributes, mainly related to the possibility of temporarily transitioning the simulation state into an incorrect configuration. At the highest classification level, we differentiate between *conservative* and *optimistic* protocols. The former guarantee that no portion of the simulation state is ever incorrect. The latter admit transient situations of incorrectness that are corrected a posteriori by reconstructing a previous correct state using state saving [32], reverse computation [5], or combinations thereof [7].

Optimistic simulations are further characterised by different levels of *aggressiveness* and *risk* [40]. Aggressiveness refers to how far forward the speculative portion of the simulation is tentatively carried out. Ideally, the speculative portion is later committed, thus improving the performance of the simulation. The concept of *risk*, on the other hand, refers to the exchange of events that were generated speculatively and may thus represent transient errors [40]. Such transient errors can cascade across processing elements and require dedicated handling as part of the rollback mechanism.

Within the spectrum of optimistic synchronisation protocols, Time Warp [20] exhibits the highest degree of aggressiveness and risk. Many variants to the Time Warp protocol have been proposed in the literature, showing that optimisation possibilities are many, as shown in Figure 1. For example, the *cancelback protocol* [21] can limit aggressiveness if certain portions of the simulation are too far from the commit horizon and the system's memory pressure is too high. In [44], a protocol is constructed that allows events to be sent only if they are guaranteed to be valid, thus being risk-free, at the cost of drastically reduced aggressiveness. This protocol was extended in [45], allowing sending only events generated by those closest to the commit horizon, to reduce the rollback probability.

As shown when comparing optimistic and conservative algorithms [4], the performance of optimistic algorithms depends heavily on the simulation model. For example, it was shown in [2] that the same model could benefit from different levels of optimism depending on its configuration. Jefferson [22] showed the feasibility and benefits of exploiting different synchronisation protocols simultaneously through mode switches at runtime. In this way, it is possible to work on the various attributes of the simulation to
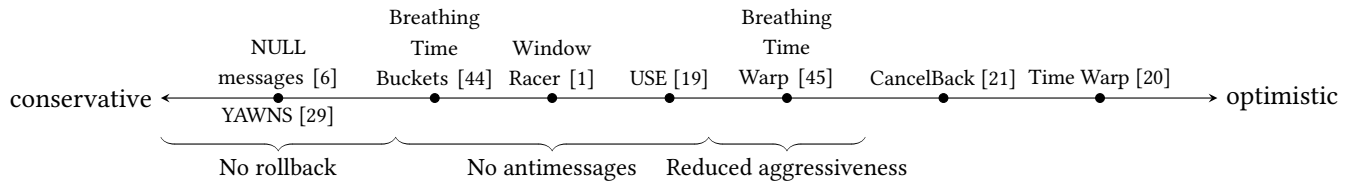
Figure 1: (Partial) Spectrum of PDES Synchronisation Protocols.

maximise performance while limiting the possible adverse effects of optimism. Jefferson's work [22] focuses on the coexistence of Time Warp with conservative protocols, thus exploiting the extremes of the spectrum of possibilities. Conversely, this paper explores the coexistence of two optimistic synchronisation algorithms within a single runtime environment. Our main contributions are two-fold:

- We present an architecture and algorithmic mechanisms to support the coexistence and seamless interaction of the asynchronous Time Warp algorithm with the recent synchronous Window Racer algorithm [1], a new window-based synchronisation algorithm.
- Experimental results demonstrate the benefits of this kind of hybrid synchronisation scheme when executing simulation models with spatially varying dynamics.

In addition, we also present a preliminary study of the potential of runtime switching between execution modes. In this context, simulations can reconfigure themselves to reduce adverse effects depending on models' runtime execution dynamics. This scenario opens up the possibility of augmenting our approach with autonomic decision policies that allow configuring at runtime the execution environment to the grain of the individual simulation entity. While a full cost model to inform run-time algorithm selections at runtime will be the subject of future work, Window Racer has been shown to outperform Time Warp in model configurations where events frequently occur with very short delays [1].

Our preliminary results show the potential of this approach in terms of increased performance and simulation efficiency. Our approach can also control *thrashing* phenomena, often observed with particular workloads in simulations based on Time Warp [43]. This is of value to modellers, allowing them to take advantage of PDES without worrying about the execution modes supported by runtime environments. In this way, the modeller can focus their work on defining the model, effectively delegating the cost of supporting the execution to the runtime environment, which is known not to always be possible in the context of PDES simulations [30].

The rest of this paper is structured as follows. In Section 2 we overview the two synchronisation protocols we fuse in our proposal. Section 3 discusses related work. Our hybrid synchronisation protocol and the supporting architecture are presented in Section 4. Section 5 presents the experimental assessment of our proposal.

2 BACKGROUND

2.1 Time Warp

The Time Warp synchronisation protocol [20] is used in PDES to achieve *eventual event safety*, ensuring that the order of events is eventually consistent across all processors. The global simulation is partitioned into a set of non-overlapping entities (also named logical processes, LPs), which maintain their own local virtual time (LVT). The LVT is the time the entity believes the simulation is currently at. Different entities can have a different LVT value at the same wall-clock time instant.

The Time Warp protocol is considered optimistic because it allows entities to proceed with their simulation even if they do not have all the necessary information, i.e. even if all required events have not yet been received. This is a concept also known as aggressiveness. Entities may send messages to one another to exchange information, but they are not blocked from proceeding with the simulation. This can lead to increased performance but also increases the probability of causality violations. Any actions taken in error can be undone using rollbacks to restore consistency. If an entity discovers that it processed events in the wrong order, it can undo its actions and restore the simulation to a previous state. This allows the simulation to continue from a consistent point rather than allowing errors to propagate.

A key concept in Time Warp is the Global Virtual Time (GVT), which acts as a common clock that all simulation entities share in the simulation. It represents the minimum of all the local LVT across all entities and in-transit messages [12]. GVT helps ensure consistency in the simulation by allowing entities to identify which events cannot be undone by any straggler event. In addition, GVT plays a crucial role in garbage collection activities. Time Warp requires maintaining information related to already-processed events and past simulation states, as long as they belong to the speculative part of the simulation trajectory. Buffers associated with a timestamp before the GVT can be safely garbage collected. The GVT also discriminates tentative actions, such as producing output or materialising errors. Its computation is time-consuming and can impact the overall simulation performance. As we will show, integrating different speculative synchronisation protocols can also be a source of optimisation to the GVT computation.

2.2 Window Racer

Window Racer (WR) [1] is a recent synchronous optimistic synchronisation algorithm for shared-memory architectures. Due to its cautious form of optimism, it is well-suited for models in which state transitions are tightly coupled across entities and frequently occur with short delays, which pose challenges to Time Warp [2].

Inspired by Steinman's Breathing Time Buckets (BTB) [45], WR alternates between an execute and commit phase. As in BTB, each processing element (PE) is assigned a portion of the simulation entities. In both algorithms, at the end of the execute phase, a newly determined GVT decides which state transitions can be committed.

Algorithm 1 Main loop of the Window Racer algorithm.

```
 1: global upperBound ← +∞
 2: global lowerBound ← −∞
 3: per-thread cel ← PRIORITYQUEUE()
 4: per-thread uel ← PRIORITYQUEUE()
 5: procedure PROCESSWINDOW()
 6:     while not reached termination criterion do
 7:         do atomically:
 8:             lowerBound ← COMPUTEGLOBALMINIMUMTIMESTAMP()
 9:             upperBound ← lowerBound + τ₀
10:         while GETTIMESTAMP(EARLIESTEVENT(uel ∪ cel)) < upperBound do
12:             nextEvent ← POP(thread.uel ∪ thread.cel)
13:             LOCK(nextEvent.entity)
14:             if REGISTEREVENT(nextEvent.entity, nextEvent) then
15:                 generatedEvents ← PROCESSEVENT(nextEvent)
16:                 cel ← cel ∪ generatedEvents
17:             UNLOCK(nextEvent.entity)
18:         uel ← ∅
19:         THREADBARRIER()
20:         for each entity do
21:             if |entity.event_list| = 0
22:             or GETTIMESTAMP(LATESTSTATE(entity)) < upperBound then
23:                 entity.event_list ← ∅
24:                 entity.state_list ← ∅
25:                 continue
26:             ROLLBACK(entity, upperBound)
27:             for each event ∈ entity.event_list do
28:                 if GETGENERATIONTIME(event) < upperBound
29:                 and GETTIMESTAMP(event) ≥ upperBound then
30:                     uel ← uel ∪ event
31:             entity.event_list ← ∅
32:             entity.state_list ← ∅
33:         THREADBARRIER()
```

Algorithm 2 Entity locking and update of the window bound.

```
 1: procedure REGISTEREVENT(entity, event)
 2:     APPEND(entity.event_list)
 3:     if GETTIMESTAMP(event) ≥ upperBound then
 4:         return false
 5:     if GETTIMESTAMP(event) < LATESTCHANGETIMESTAMP(entity) then
 6:         refState ← GETEARLIERSTATE(entity, event)
 7:         newUpperBound ← GETTIMESTAMP(refState)
 8:         do atomically:
 9:             upperBound ← min(upperBound, newUpperBound)
10:         ROLLBACK(entity, GETTIMESTAMP(refState))
11:         return true
12:     SAVESTATE(entity)
13:     return true
```

However, important differences lie in the granularity and policy according to which the new GVT is determined. In BTB, the new GVT is simply the earliest timestamp of any event that crosses PE boundaries. This permits a clear delineation of the execute and commit phases, allowing the execute phase to proceed without any PE interaction. However, when simulating systems of entities that interact globally and with short delays, the resulting synchronisation windows can become exceedingly small.

WR alleviates this issue by loosening the entity-to-PE assignment in the execute phase. Algorithms 1 and 2 show WR's main loop, entity-level locking, and GVT negotiation as pseudo-code. Each PE maintains a *unconditional event list* (uel) holding events guaranteed to be committed at some point throughout the simulation, and an *conditional event list* (cel) holding events generated in the current round's execute phase, some of which may have been generated in error and may never be committed. At the beginning of the execute phase, each PE considers the local entities' events in timestamp order. However, any newly generated events are also executed, regardless of the target entity's PE assignment. Race conditions are ruled out by acquiring a lock on an event's target entity before saving the entity's state, appending the event to a per-entity event list, and executing the event. This allows PEs to execute entire chains of dependent events without handing the execution off to other PEs or separating the execution into multiple rounds.

Throughout this execution scheme, the PEs negotiate the new GVT based on entity-level straggler events. When a straggler with timestamp t is encountered, the target entity is rolled back to its latest state earlier than t, and the new GVT is updated to exclude the earliest event displaced by the straggler. Through this process, the value of the global variable holding the new GVT gradually

decreases throughout the execution phase, allowing PEs to immediately cease execution when their earliest event is past the new GVT. The algorithm's name is inspired by the PEs' "race" to fit as many events as possible into a gradually closing synchronisation window.

In the commit phase, PEs iterate through all local entities. If necessary, the entities are rolled back to the GVT, and any events from their event lists created prior to the GVT but with timestamps beyond the GVT are inserted into the PEs' unconditional event list.

As a consequence of its strictly synchronous execution scheme, Window Racer limits the deviation of threads from the GVT. Our aim in combining the algorithm with Time Warp is to limit misspeculations in tightly coupled dynamics through Window Racer's cautious form of optimism, while more loosely coupled model segments can benefit from Time Warp's aggressiveness.

3 RELATED WORK

The literature offers a wealth of work attempting to control or exploit aggressiveness and risk in speculative simulations. The first class of proposals is related to the throttling of simulation entities straying too far from the commit horizon. In this class, proposals such as *Bounded Time Warp* [47] and *window-based throttling* [39] limit optimistic processing by defining a static optimism window. The main problem with these proposals is the possibility that different simulation models (or even individual entities in the same model) may require different window sizes for optimal performance. An incorrectly sized window can drastically reduce performance. *Adaptive Time Warp* [3] controls optimism by forcing a simulation entity that experiences many rollbacks to freeze for a time *BW*. Determining *BW* is complex, often leading to configurations that may not maximise speedup. Furthermore, as discussed in [39, 47], the value of *BW* may be different for each simulation entity.

Penalty Based Throttling [39] activates the entities that receive the fewest antimessages, associating them with a penalty that captures the number of antimessages received. In a complex simulation, all simulation entities can have a non-negative penalty at a particular time instant, leading to an excessive reduction in aggressiveness.

A second class of proposals uses the concept of dynamically adapted *windows* to determine which events can be executed while limiting the incidence of rollbacks. *Breathing Time Warp* [45] combines *Breathing Time Buckets* [44] and Time Warp. While Breathing Time Buckets only allows events to be sent when they are valid (thus with very low aggressiveness), Breathing Time Warp is based on the principle that events closer to the GVT have a lower probability of

being cancelled. Therefore, N_1 events close to the GVT are optimistically executed, and N_2 events after that point are executed using *Breathing Time Buckets.Adaptive Bounded Time Windows* [31] uses the concept of *useful work*. Windows are sized to maximise speed. *Adaptive Time-Ceiling* [26] is based on a similar concept, although window sizes are chosen from a set of discrete values. The hybrid synchronisation algorithm proposed in the present paper is related to Breathing Time Warp in its combination of a window-based optimistic synchronization algorithm with Time Warp. However, we take a fundamentally different approach: while Breathing Time Warp combines two algorithms in *time* by alternating between the two, we combine these algorithms in *space*, i.e., different pools of simulation entities are handled by the two algorithms, which coexist and interact. The architectural and algorithmic considerations required to facilitate this coexistence comprise the main contributions of our paper.

A third class of works deals with aggressiveness and risk employing scheduling policies. After the seminal work in [23] has shown that Time Warp simulations are conservative optimal, although subject to stragglers that can still hamper performance, many proposals have dealt with heuristics to schedule entities based on their rollback behaviour and productive work. *Useful work* [33] is a performance index based on control theory that enables scheduling policies to control the optimism of a time warp simulation, reduce the rollback frequency, and reduce memory usage and wasted lookahead computation as a secondary effect. *Approximate time* [14] is a partial order of events that allows the exploitation of temporal uncertainty to reduce the rollback probability. Events are not associated with a single timestamp but rather with an interval. Overlapping events can be reordered to provide equivalent schedules to avoid executing a rollback operation upon receiving a straggler message. A similar objective is pursued through *symbolic execution* [49], in which a group of uncertain cases are jointly simulated in the same run to explore possible configuration parameter intervals. *Aggressiveness/Risk Effects-based Scheduling* (ARES) [8] controls optimism by scheduling with a higher priority the simulation entities whose next event has a lower probability of being eventually rolled back. This is done by selecting a set of candidate simulation entities with a low probability of being eventually undone; then, it chooses an entity among the candidates to minimise the number of new events to be notified. *Share-Everything PDES* [19] exploits a short-framed temporary binding between simulation entities and worker threads. The proposal exploits a per-node global future event set that can be concurrently exploited to determine the next event closest to the commit horizon. This approach can reduce the incidence of rollbacks and the generation of stragglers thanks to its controlled aggressiveness.

In some works, *hybrid approaches* have been proposed that add optimism to conservative algorithms. SRADS with local rollback [10] and speculative computing [28] optimistically process unsafe events locally, thus confining rollbacks to local entities. Breathing Time Buckets [44] is similar to SRADS but can dynamically vary the conservative time windows based on global and local event horizons. This technique performs poorly under small lookahead conditions, and the GVT must be computed with no in-transit messages. *Bounded lag restriction* [24] uses the measure of the minimum distance between entities to decide safe events based

on programmer-provided apriori lower bounds between causally related events. The *rollback relaxation* [50] and *unsynchronised parallel simulation* techniques [38] relax causality constraints for memoryless logical processes or completely ignore causality violations for queueing models to reduce the rollback overhead. The final results may be imprecise, and in general, these techniques cannot be applied to all classes of simulation models.

Also, proposals related to *load sharing* and *load balancing*, such as [17, 26, 31, 48] can reduce the number of rollbacks, thanks to improved exploitation of local synchronisation based on smallest-timestamp first scheduling. These proposals have performance in mind and implicitly try to limit optimism globally. Nevertheless, secondary effects should be dealt with explicitly.

In general, all these classes of proposals are generally still bound to a single family of optimistic execution. They provide performance improvements under specific workloads and can cope well with simulation scenarios that show characteristic behaviour of the parameters and aspects taken into account when the algorithms, even adaptive ones, were designed. Nevertheless, the degree of adaptivity that they can handle could be improved. In this paper, we overcome this general limitation by showing how different scheduling and synchronisation schemes with different levels of aggressiveness and with or without risk can be combined. Even if we only combine two different algorithms, the architecture is general and can be extended to include additional ones. Furthermore, we show experimentally how our proposal can effectively handle a challenging class of models for traditional optimistic PDES simulation since the dynamics change significantly within the same model due to configuration parameters [2].

A proposal sharing our ultimate goal is *Virtual Time III* [22], where a unification framework between conservative and optimistic synchronisation is proposed. This framework considers conservative algorithms as *accelerators* to Time Warp, given the non-existent overhead introduced to forward execution. Therefore, Virtual Time III enables some simulation entities to execute conservatively, while others execute optimistically, at the same time. In this context, some entities are subject to throttling due to the execution in conservative mode. There are three differences between Virtual Time III and Time Racer. First, we highlight the benefits of mixing multiple optimistic protocols to exploit aggressiveness and risk to different extents in the same simulation run. Second, we allow a group of worker threads to run cooperatively groups of simulation entities (with an $M : N$ mapping), while Virtual Time III still considers the traditional $1 : N$ mapping between worker threads and entities. Last, we provide a reference implementation and experimental assessment, which is lacking in [22].

4 HYBRID SPECULATIVE SYNCHRONISATION

The Window Racer (WR) algorithm presented in [1] uses n threads to carry out the simulation of k simulation entities cooperatively. Conversely, in the classical Time Warp (TW) implementation [20], there is a binding between a worker thread and a set of simulation entities—this binding can be fixed or temporary, as in the case of load-balancing policies such as those presented in [17, 42, 48].

Let us consider for now a single simulation node in which m cores are used to perform processing and housekeeping tasks. Allowing

the coexistence of different synchronisation algorithms, such as WR and TW, requires that a number $n \leq m$ of threads, at a given instant in time, performs the tasks required by the WR algorithm, while the remaining $m - n$ threads execute TW's activities.

Given that both synchronisation algorithms can execute a complete simulation alone, the integration is trivial if there is no interaction between the simulation entities managed by either algorithm. Conversely, as soon as a simulation entity schedules an event to another entity handled by a thread running the other algorithm—we name it a *cross-algorithm* interaction—, care must be taken to ensure this interaction does not create any inconsistency in the speculative simulation trajectory. In the following, we describe the methodology to support a correct and efficient integration between the two synchronisation algorithms for cross-algorithm interactions.

4.1 Cross-Algorithm Priority Inversion

A cross-algorithm event scheduling might require reconstructing a previous simulation state if the event is a straggler for the recipient. If a TW thread manages the destination simulation entity, the scenario poses no harm: a traditional rollback operation can restore the previous consistent state from which to restart the execution. Conversely, rollbacks are exclusively local to a window in the original WR algorithm. Therefore, when the processing of a window is completed, the events associated with that window are immediately committed—WR does not need a dedicated algorithm to calculate the Global Virtual Time (GVT) as in the case of TW. A straggler received by a WR thread from a TW one requires additional information to reconstruct a previous consistent state.

When a thread running in WR mode receives a straggler message, there are two cases to consider. In the first case (see Figure 2a), the straggler hits before the beginning of the current or a previous window. In this case, the solution we adopt considers the entire WR window as an *atomic unit of execution*. All events are undone, and the simulation restarts from a previous window.

In the second case, the straggler falls within the current window (see Figure 2b). In this case, there is no need to flush the entire window. Indeed, all the events executed before the straggler are still (speculatively) correct. Therefore, in this case, we simply update the window's upper bound, closing the window at the straggler's timestamp. All inconsistent events will be undone.

To support the invalidation of an entire window, we keep checkpoints also for previous windows, which was unneeded in the original WR algorithm. As a first approximation, to support the cancellation of a window, we could take a snapshot of all simulation entities associated with the n threads executing in WR mode before a new window starts. This approach achieves correct execution: should a straggler be received, window cancellation can be supported by restoring the state of all entities involved. However, it may be sub-optimal for several reasons. First, windows may be extremely short, thus requiring many checkpoints. Indeed, as shown in [2], WR performance may suffer if simulation entities are significant numbers and have a very high message exchange rate.

In the Time Warp literature, however, the problem of selecting a checkpointing interval appropriate to the dynamics of the model has been extensively addressed (e.g., in [2, 11]). These techniques generally involve the possibility of simulation model states being organised in arbitrarily complex data structures, e.g. based on the use of dynamic memory, as in [9, 35]. Therefore, in our integration, the management of checkpoints is entrusted to an *autonomic checkpoint manager* [36] that determines, during the execution of the simulation, which is the most convenient time instant for capturing a snapshot of the simulation state of an entity. This strategy brings about an important change to the WR algorithm: in this way, the need to capture a snapshot before the execution of each event disappears. However, in this way, it is necessary to introduce the execution of a *coasting forward* phase in WR as well to allow the realignment of a simulation state if a rollback occurs. This strategy applies also to window-local rollbacks.

Therefore, it is necessary to properly manage the simulation's *Past Event Set* (PES) to properly allow the reprocessing of events if a rollback occurs and a consistent previous simulation state needs to be reconstructed. Unlike the *Future Event Set* (FES), the PES needs to maintain *per-entity* information, as a rollback affects one single entity. Therefore, a different PES is used for each entity. As a data structure, our integration relies on a doubly-linked list for the PES. The use of this data structure is different from the FES since its purpose is to support the efficient execution of coasting forward. This operation is linear by nature: once the first event to be reprocessed by a particular simulation entity has been identified, all subsequent events must be reprocessed in order. Hence, the advantage of using a list of events for each simulation entity.

An important aspect of our integration concerns the calculation of the GVT. Since the simulation involves threads executing the TW algorithm, it is not possible to disregard classical GVT calculation algorithms that determine a lower bound of the real GVT value (e.g., [16, 27, 34, 46]). However, in this GVT calculation, inspecting the PES of all entities is not necessary. Indeed, it is possible to exploit the concept of window atomicity: all entities managed by WR threads enjoy the automatic GVT reduction inherent in the WR algorithm. Therefore, it will be necessary to calculate the GVT reduction between the logical times of all entities managed according to the TW scheme and the initial timestamp of the last correctly processed WR window.

4.2 Event Generation and Scheduling

Another aspect to deal with in integrating the algorithms is managing the generation of new events and their scheduling activities. In particular, given the different aggressiveness of the TW and WR algorithms, we have decided to organise FES differently for the threads running the two. For both algorithms, FESs are priority queues implemented using k-heaps, as they have experimentally shown good performance in the case of disparate workloads [37]. In particular, by using a k-heap, extracting the next event to be executed has a constant cost, which is particularly important in the case of the n threads executing the WR algorithm.

Indeed, for the n WR threads, a *single shared FES* is used. While this choice requires access synchronisation (e.g. through spinlocks), the advantage lies in that the n threads can effectively cooperate in the execution of events that fall within the various windows. Conversely, the TW threads use a single FES for all the associated simulation entities. This strategy deviates from classical solutions
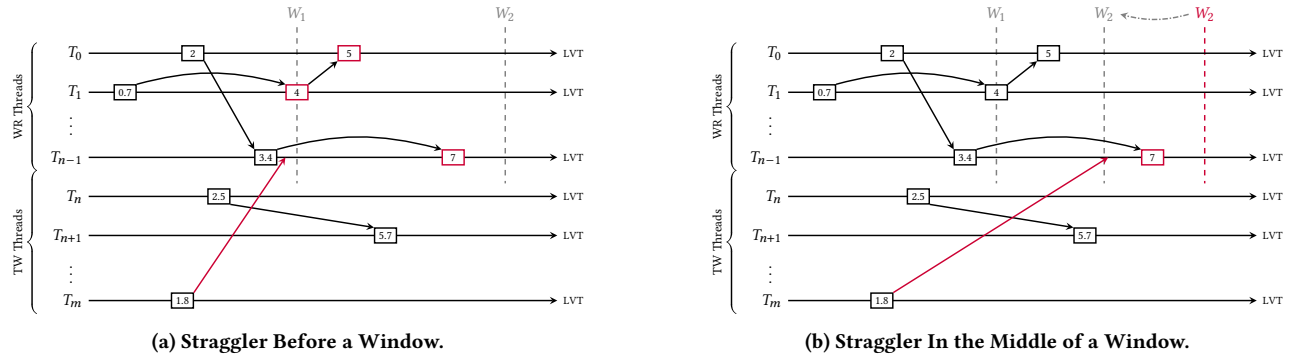
(a) Straggler Before a Window.   (b) Straggler In the Middle of a Window.

Figure 2: Straggler Messages Invalidating Window Racer Windows.

in the literature, in which a single queue is provided for each simulation entity. However, as shown in [37], fewer queues can lead to non-negligible performance benefits. The lack of a per-simulation entity FES requires determining, upon event schedule, what is the proper FES to insert the newly-generated event into. We support this mapping by using a hash table that associates the unique id of an entity with the FES that maintains the events to be processed.

During the execution of events, newly-generated events are handled differently depending on their source, destination, and timestamp. Events generated by TW threads are immediately delivered to the relevant FES—if they are stragglers, they will cause a rollback. If they fall into the current WR window, they determine an update of the upper bound of the window—this is the scenario we already depicted in Figure 2b. Conversely, all events generated by a WR thread are inserted in a per-thread *output queue*.

Logically, this output queue has a dual purpose. On the one hand, it buffers events that could be undone should a local rollback to the window cancel the processing of the generated event. On the other hand, it maintains causality information for all those events sent to TW threads: in the event of a window rollback, all those messages will have to be cancelled with anti-messages.

At the end of the execution of the window, all entity events handled by WR threads in the output queue are placed in the cooperative FES of WR threads. Other messages are retained in the output queue until the GVT overtakes the entire window. At that point, the traditional *fossil collection* operation [20] allows the memory buffers to be retrieved. This dualistic use of the output queue allows the different degrees of aggressiveness, proper of the two WR and TW algorithms, to be handled correctly.

Overall, the operations carried out by WR threads are reported in Algorithm 3. The global FES (line 2) is shared among all worker threads cooperating to process the current window (line 3). The WR threads maintain a per-thread output queue (line 4) that is used in conjunction with the FES to determine when the processing of the current window is over and what is the next event to schedule (lines 5–10). As discussed, the output queue is also used to keep track of all the generated events (line 22, 24).

Given the cooperative nature of WR, we must ensure that a single simulation entity is not concurrently executed by two different worker threads. To this end, we employ a locking mechanism based on atomic read-modify-write instructions that ensure that only a

single worker thread will take care of an entity if two events are extracted concurrently—lines 12, 14, 23.

If a straggler message is received, the worker thread managing the entity hit by the straggler will reduce the window size (lines 13–19). The update of the upper bound must be done atomically, because multiple threads may be managing stragglers at the same time. In the case of concurrent update, the minimum among all the new tentative values should be stored. Therefore, we rely on a Compare-and-Swap based retry loop, thus implementing a non-blocking update.

The window is completely processed when the next event to be processed is scheduled at a timestamp beyond the window's upper bound (line 5). At this point, the threads should deliver the events still present in the output queue to the FES (lines 27–31).

4.3 Dynamic execution mode switching

An important aspect of managing a hybrid synchronisation mechanism such as the one proposed in this work is the possibility of dynamically switching from one execution mode to another. Indeed, as was shown in [2], depending on the dynamics of the simulation model, different synchronisation modes may prove to be successful. Clearly, in a general simulation, it is possible for these dynamics to change, just as it is possible for different parts of the model to behave differently. Therefore, to maximise performance, it is desirable that the number n of threads running in a given mode changes during the same simulation.

The different nature of the two synchronisation algorithms considered in this work requires certain precautions to make this transition effective. Let us first consider the simplest case in which a thread executing in TW mode must switch to WR mode. The organisation of FESs described above requires that, in the transition, a TW thread inserts all future events of its FES into the one shared between all threads executing in WR mode.

Transitioning from TW to WR mode must be performed with minimal invasiveness to WR execution. If a TW thread merely modifies the FES of WR threads, artificial rollbacks may be introduced. Conversely, WR's windowed nature allows exploiting the upper bound to discriminate between messages that may or may not generate a rollback. The thread that wants to migrate from TW to WR execution, therefore, can adopt the scheme shown in Algorithm 4.

Algorithm 3 Window Management Algorithm

```
 1: global windowUpperBound ← +∞                                                    ▷ Visible to all worker threads
 2: global pastWindows ← STACK( )                                                       ▷ Past committed windows
 3: global FES_WR ← PRIORITYQUEUE( )                                            ▷ Window Racer Future Event Set
 4: per-thread FES_TW ← PRIORITYQUEUE( )                                          ▷ Time Warp Future Event Set
 5: global rollbackWindow ← false                                             ▷ If set, an older window will be restored
 6: procedure PROCESSWINDOW( )
 7:     outputQ ← PRIORITYQUEUE( )                                        ▷ Events generated in the current window
 8:     while NEXT(FES_WR) < windowUpperBound OR NEXT(outputQ) < windowUpperBound do    ▷ FES_WR is accessed atomically
 9:         if NEXT(outputQ) < NEXT(FES_WR) then
10:             nextEvent ←POP(outputQ)
11:         else
12:             nextEvent ← POP(FES_WR)
13:         simEntity ← GETENTITYOF(nextEvent)
14:         LOCK(simEntity)                                   ▷ Mark the simulation entity as being processed by a WR thread
15:         if GETTIMESTAMP(simEntity.lastProcessedEvent) > GETTIMESTAMP(nextEvent.time) then         ▷ Straggler detected
16:             UNLOCK(simEntity)
17:             if GETORIGINPARTITION(event) = TW then
18:                 rollbackWindow ← true
19:             do atomically:
20:                 if windowUpperBound > GETTIMESTAMP(nextEvent) then
21:                     windowUpperBound ← GETTIMESTAMP(nextEvent)
23:             break
24:         generatedEvents ← PROCESSEVENT(nextEvent)                    ▷ Calls the model event handler and returns the generated events
25:         UNLOCK(simEntity)
26:         outputQ ← outputQ ∪ generatedEvents
27:     THREADBARRIER( )                                                                      ▷ Window is over
28:     if rollbackWindow then
29:         outputQ ← ∅
30:         do
31:             windowToRestore ← POP(pastWindows)
32:         while windowToRestore ≥ windowUpperBound
33:         for each simEntity do
34:             ROLLBACK(simEntity, windowToRestore)
35:         THREADBARRIER( )
36:         rollbackWindow ← false
37:         return
38:     pastWindows ← pastWindows ∪ {windowUpperBound}
39:     for each event ∈ outputQ s.t. GETGENERATIONTIME(event) < windowUpperBound do
40:         if GETDESTINATIONPARTITION(event) = TW then
41:             FES_TW ← FES_TW ∪ {event}
42:         else
43:             FES_WR ← FES_WR ∪ {event}
44:     outputQ ← ∅
45:     for each entity do
46:         ROLLBACK(simEntity, windowUpperBound)
47:         FOSSILCOLLECTION(simEntity)
48:     THREADBARRIER( )
```

Initially, the migrating thread signals the start of the transition, forcing the WR threads to wait at the end of the current window. It then checks the upper bound value of the current window. If this value is less than the time of the next event in its FES, all the events to be processed belong to the *next* window. Therefore, the thread moves its events into the WR's FES.

Conversely, if its next event has a timestamp less than the upper bound, we are in the scenario of Figure 2. Here, if the thread were to insert new events into the WR queue, it could generate a priority inversion concerning the activities of the n threads executing in WR mode. Therefore, the migrating TW thread appropriately exploits the aggressiveness of Time Warp, as shown in Algorithm 4: it will continue to process events in the FES until the logical time of the next event does not exceed the upper bound of the current window. The procedure's termination is guaranteed by the signalling flag forcing the WR threads to wait for the conclusion of the transition.

This migration approach could lead to high costs if handled incorrectly. Indeed, WR threads may have to wait a long time for the migrating TW thread to complete its realignment. Therefore, when choosing a thread to switch from TW to WR mode, selecting a thread further along in logical time than the current window is crucial. If no such thread exists, the one with the next event closest in logical time to the end of the current window should be chosen. This aspect, as well as the choice of when to make the transition and the number of TW threads involved, requires the definition of an *autonomic policy* outside this work's scope.

Switching from WR execution mode to TW requires more care. Threads executing in WR mode have no inherently bound simulation entities: events associated with a given entity can be cooperatively executed by multiple threads in alternation. If a WR thread is to turn into a TW thread, it becomes necessary to manage the output queue discussed above appropriately. Including all current messages in the output queue in the various FESs would not be consistent with its handling as described above. Conversely, we place all events destined for an entity running on TW threads (including the thread performing the mode change) in the corresponding FESs.

Algorithm 4 Mode Switch from TW to WR

```
global need_switch
global switching_thread_id
global warp_threads_count
global racer_threads_count
procedure DoSwitch()
    if NOT need_switch then
        return
    if ThreadType() = WARP OR
      NOT ThreadId() = switching_thread_id then
        return
    ThreadBarrier()
    if ThreadId() = switching_thread_id then
        if ThreadType() = WARP then
            warp_threads_count ← warp_threads_count − 1
            racer_threads_count ← racer_threads_count + 1
            SetThreadType(RACER)
        else
            warp_threads_count ← warp_threads_count + 1
            racer_threads_count ← racer_threads_count − 1
            SetThreadType(WARP)
        need_switch ← False
    ThreadBarrier()
```

On the other hand, the output queue is assigned to a WR thread, which will carry on its management in a manner consistent with what is described in Section 4.2.

4.4 Going Distributed

Using our hybrid synchronisation scheme in a distributed setup is straightforward. WR is designed for execution in shared memory, whereas TW is inherently capable of handling the causality violations that may arise from a distributed execution. Therefore, the hybrid scheme we have described can be immediately used on distributed deployments due to the presence of TW.

Considering windows as atomic processing units ensures that, unlike the original proposal in [1], if a straggler message is received prior to a window, it will be cancelled entirely.

In a distributed deployment, therefore, the fact that there are multiple concurrent instances of the WR algorithm running on multiple nodes does not require making these instances aware of the presence of the others. If WR threads receive a straggler message, they will work together to restore an earlier state of the window affected by the straggler, resuming execution.

5 EXPERIMENTAL ASSESSMENT

In this section, we detail the setup and the results of the experimental assessment we carried out for our proposal. We have not considered distributed simulations due to the assumption of shared memory in the WR algorithm. As mentioned in Section 4.4, the execution dynamics in a distributed setup would heavily rely on the TW algorithm, thus subsuming results already known in the Time Warp literature.

As a first testbed application, we have used the classical PHold benchmark [13], a synthetic model that creates a series of events exchanged between simulation entities. It allows controlling simulation parameters such as the number of entities, events' delay, and workload distribution across the overall simulation. The second benchmark we used is a simulation model of the epidemic spread of contagion [2], an extension of the traditional agent-based formulation of the susceptible-infected-recovered (SIR) model [25].

Our analysis was conducted using a machine equipped with two AMD® EPYC™ 7452 processors @ 2.9 GHz, each consisting of 32 physical cores and 64 hyperthreads, for a total of 64 physical cores and 128 hyperthreads—hyperthreads were turned off in the experimental assessment. The machine is equipped with 256 GB of RAM. All experimental results are averaged over 20 different runs.

5.1 Testbed Applications Configuration

For PHold, we have simulated a total of 131,072 simulation entities mapped to a variable number of threads in multiple runs. The entities are divided into a *high-* and a *low-activity* partition, the former composed of 64 entities and the latter of the remaining 131,008. With a 50% probability, an entity randomly selects a destination for an event in the high-activity partition. This way, we can mimic a scenario where a part of the simulation has a higher load.

Depending on the number of threads, there could be a significant skew on the logical clocks of the entities that could increase the rollback probability. The high-activity partition suffers most from rollbacks because of the denser concentration of events that a single straggler can undo.

We considered the SIRS variant for the epidemiologic model, where recovered agents eventually revert to the vulnerable state. Each agent is located in one of an adjustable number of fully connected domains, each with the same initial number of agents. Agents have eight randomly selected neighbours in the same area, so the number of areas affects how localised agent interactions are.

Transition delays are drawn from exponential distributions with fixed or dynamic rates. The infection rate for susceptible agents is proportional to the number of infected neighbours. As a result, agents entering or leaving the infected state must notify their neighbours to reschedule their transition to the infected state based on the new rate. Transitions from the recovered state to the susceptible state occur at a constant rate of 1. Two other transitions introduce dynamic changes to the topology defined by the neighbourhood relationships of the agents. The first type of transition randomly changes an agent's neighbours within its current region, potentially changing its infection rate or the infection rates of its neighbours. The second type of transition involves the movement of an agent.

The second type of transition randomly moves an agent to another region and connects the agent to new neighbours in the new region. The rates at which these two types of transition occur allow us to control the degree of computational load and agent interaction within each region, as well as the interdependence of transitions between regions. Overall, this system is similar to epidemic models used in real-world epidemic studies [18], which attempt to capture the effects of daily and long-distance mobility of populations.

5.2 Experimental Results

In Figure 3, we show the speedup over the corresponding sequential simulation for the PHold benchmark, using a total of 24 and 32 threads empowering the two integrated algorithms. The plots depict the performance variation as the number of threads responsible for simulating the high-activity partition changes. The two curves correspond to the high-activity partition simulated by the TW or WR threads. In some configurations, simulation runs are more than 10 times slower than the corresponding serial execution. The dashed
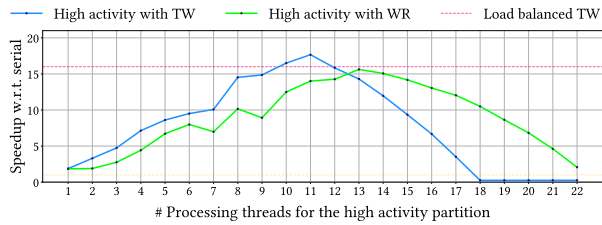
(a) 24 threads
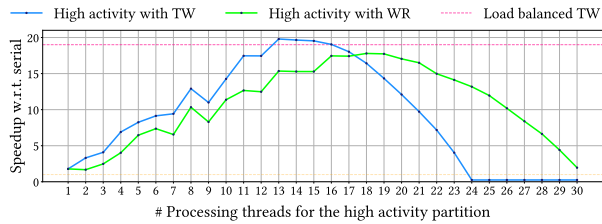
(b) 32 threads

Figure 3: PHold Performance

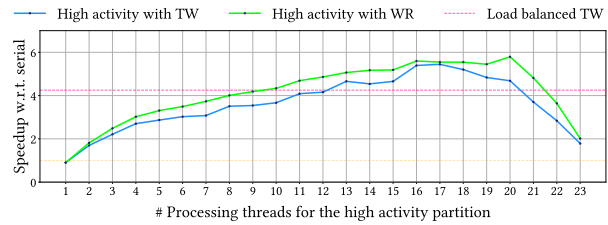**(a) 24 threads**

(b) 32 threads

Figure 4: SIR Performance

curve represents a configuration where the high-activity LPs are uniformly distributed among processing threads to balance the computational load. With our knowledge of the chosen models, this can be accomplished statically and accurately.

In the imbalanced model, the total execution time changes drastically depending on the workload partitioning to the threads. Noteworthy, when using TW and WR for the high-activity partition, a single configuration exhibits the best performance, independently of the total thread count. A less apparent observation is that the optimally partitioned configurations surpass the load-balanced configuration in performance. This is because, despite achieving better load balance, the communication of the high-activity partition becomes dispersed into more threads, resulting in less-efficient event management operations.

The important result is that, with PHold, the best static configuration is always found using only TW threads. Nevertheless, thrashing phenomena are observed for the TW algorithm if the thread count is too high. This is because when fewer LPs are bound to a TW thread, the system becomes over-optimistic, and the high-activity partition is subject to more rollbacks. The rollback cost is much higher for the high-activity partition, as more work is wasted. In contrast, the WR algorithm is less sensitive to increased concurrency in the high-activity partition because WR threads can effectively leverage the reduced aggressiveness to decrease the rollback occurrence in the simulation. Apparently, WR threads are more resilient to tighter simulation interactions, but their average throughput is lower. This is evidenced by configurations equipped with WR achieving peak performance with more threads.

Figure 4 reports the SIR-model speedup for the same thread configurations. As can be seen, the scenario is significantly different: we consistently achieve improved performance over the sequential simulation when WR threads manage the high-activity partition. This is because the cost of rollbacks is higher, and the LP interactions in the high-activity partition are highly dynamic and tightly
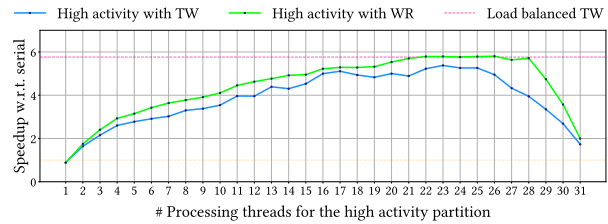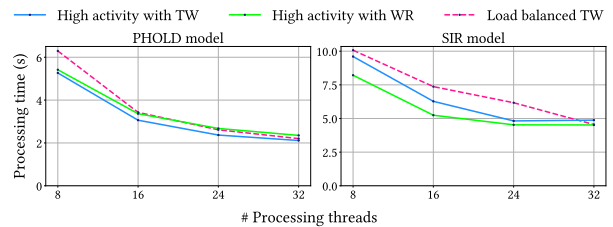

**Figure 5: Strong scaling, picking the best partitioning**

coupled. In this scenario, TW becomes excessively optimistic, causing significant clock skewing and leading to worse performance. The more cautious approach of WR can achieve better returns from optimism, reducing the overall number of rollbacks.

In summary, Figure 5 demonstrates that for the PHOLD model, the partitioned TW configurations yielded the best performance, whereas, in the SIR model, the hybrid configurations performed the best. However, with 32 threads, all configurations were more or less on par. The SIR model has a high degree of coupling, so even with 24 threads, we achieved a speedup comparable to what was attained with 16 threads. This indicates that we were nearing the parallelisation limit for the model, at least using our current techniques. Interestingly, the load-balanced TW compensated for its inefficiencies through sheer processing power. Nevertheless, we can deduce that the hybrid approach is considerably more efficient for the SIR model, mainly when using fewer threads.

To provide further insight into these findings, we have illustrated the total number of rollbacks experienced by the TW and WR thread pools throughout the simulation in Figure 6. The results show that the WR configuration experiences significantly fewer rollbacks overall. This is due to the algorithm's avoidance of over-optimism, resulting in a lower probability of a WR thread engaging in an
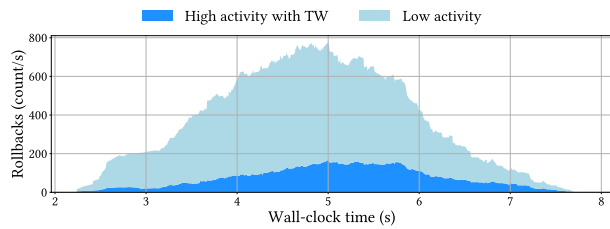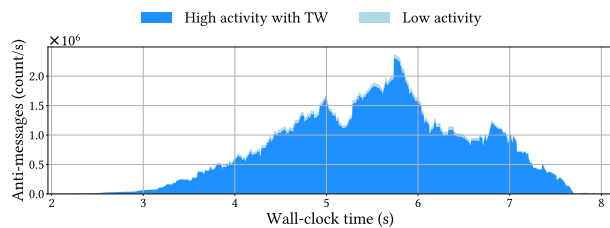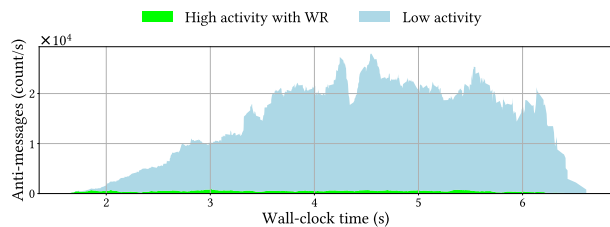
(a) Best TW-only configuration
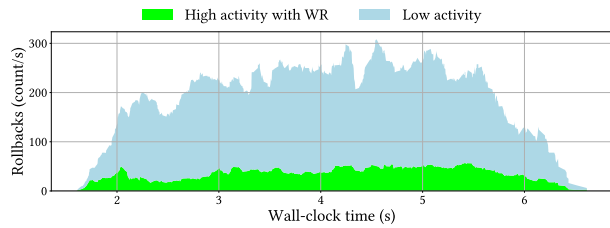
(b) Best hybrid configuration

Figure 6: SIR Rollbacks over time: 24 threads

(a) Best TW-only configuration

(b) Best hybrid configuration

Figure 7: SIR Antimessages sent over time: 24 threads

incorrect speculative trajectory. In contrast, the configuration using only TW threads experiences a much higher rate of rollbacks.

Figure 7 demonstrates that the high-activity partition managed by TW is responsible for the increased number of rollbacks in TW-only configurations. In contrast, the very same partition managed by WR generates a minimal amount of antimessages. However, the WR threads must wait at the end of windows, leading to blocking synchronisation when committing new windows. Therefore, considering the better performance provided by the hybrid configuration, it can be inferred that the time spent by TW threads in incorrect trajectories and sending antimessages is superior but roughly comparable to the time spent by WR in blocking wait.
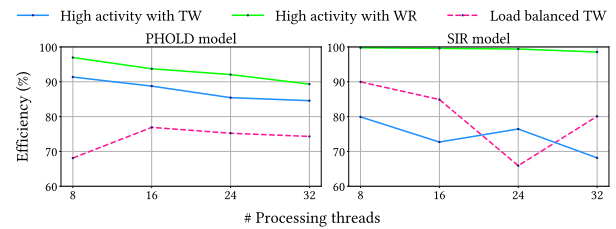

Figure 8: Efficiency, picking the best partitioning

The last finding is validated by the simulation efficiency shown in Figure 8, demonstrating that the hybrid approach outperforms the other strategies for both the evaluated models. Efficiency is determined by computing the proportion of committed events in relation to the overall number of executed events, expressed as a percentage. It is once again evident that the WR's less aggressive characteristics significantly decrease rollback occurrences, albeit at the expense of idle periods on processing threads. In particular, in a model such as SIR, with a limited degree of parallelism that can be captured in the high-activity portion of the model, the hybrid synchronisation method can achieve remarkably high levels of efficiency.

These plots over time show that, even for a simplified epidemic model, we observe a dynamicity in behaviour that an autonomic policy can potentially exploit. From this overall experimentation, we can draw the following observations. Evenly distributing the computational workload among threads in a uniformly synchronised simulation may not always be optimal, as communication costs, even within the same machine, can influence its effectiveness. Partitioning the simulation in uneven ways can bring measurable performance benefits and allows using different synchronisation algorithms that better adapt to the model's dynamics.

We have shown that WR and TW can adequately capture the model's parallelism under different conditions. In all cases, there is an optimal static configuration that depends on the model characteristics. Since the workload dynamics can change over time, we emphasise that this optimal static configuration may also vary. Therefore, the autonomic policy we envisaged in this paper is fundamental because it can capture the best-suited parallelism level in certain simulation phases, and tune the configuration to deliver better performance.

6 CONCLUSIONS AND FUTURE WORK

We have presented the methodology to integrate different synchronisation protocols for speculative PDES. Experimental results clearly demand the introduction of an autonomic policy for the proper selection of the number of threads running the WR and the TW algorithms. An additional dimension of optimisation could entail dealing with multiple pools of WR threads, thus exploring how multiple FESs for the WR part may affect the simulation performance. Also, by introducing lookahead information, we could include conservative algorithms, thus effectively providing an implementation for a multi-modal hybrid simulation engine.

All these aspects will be the subject of future-work investigation.

REFERENCES

[1] Philipp Andelfinger, Till Köster, and Adelinde Uhrmacher. 2023. Zero Lookahead? Zero Problem. The Window Racer Algorithm. In *Proceedings of the 2023 SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '23)*. ACM, New York, NY, USA, 12 pages.

[2] Philipp Andelfinger, Andrea Piccione, Alessandro Pellegrini, and Adelinde Uhrmacher. 2022. Comparing Speculative Synchronization Algorithms for Continuous-Time Agent-Based Simulations. In *Proceedings of the 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT '22)*. IEEE, Piscataway, NJ, USA, 57–66. https://doi.org/10.1109/DS-RT55542.2022.9932067

[3] Duane Ball and Susan Hoyt. 1990. The Adaptive Time-Warp Concurrency Control Algorithm. In *Distributed Simulation (PADS '90)*, David Nicol (Ed.). Society for Computer Simulation, San Diego, CA, USA, 174–177.

[4] Christopher D Carothers and Kalyan S Perumalla. 2010. On Deciding Between Conservative and Optimistic Approaches on Massively Parallel Platforms. In *Proceedings of the 2010 Winter Simulation Conference*, Björn Johansson, Sanjay Jain, and Jairo Montoya-Torres (Eds.). IEEE, Piscataway, NJ, USA, 678–687. https://doi.org/10.1109/WSC.2010.5679119

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

[6] Kanianthra Mani Chandy and Jaydev Misra. 1979. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering* SE-5, 5 (Sept. 1979), 440–452. https://doi.org/10.1109/tse.1979.230182

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

[8] Vittorio Cortellessa and Francesco Quaglia. 2000. Aggressiveness/Risk Effects Based Scheduling in Time Warp. In *Proceedings of the 2000 Winter Simulation Conference*, Jeffrey A Joines, Russel R Barton, Keebom Kang, and Paul A Fishwick (Eds.). IEEE, Piascatawy, NJ, USA, 409–417. https://doi.org/10.1109/WSC.2000.899746

[9] Samir Das, Richard M Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. 1994. GTW: A Time Warp System for Shared Memory Multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference*, Jeffrey D Tew, Mani S Manivannan, Deborah A Sadowski, and Andrew F Seila (Eds.). Society for Computer Simulation International, San Diego, CA, USA, 1332–1339. https://doi.org/10.1109/WSC.1994.717527

[10] Phillip M Dickens and Paul F Reynolds, Jr. 1990. SRADS with Local Rollback. In *Distributed Simulation (PADS '90)*, David Nicol (Ed.). Society for Computer Simulation, San Diego, CA, USA, 161–164.

[11] Josef Fleischmann and Philip A Wilsey. 1995. Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS '95)*. IEEE Computer Society, Piscataway, NJ, USA, 50–58. https://doi.org/10.1145/214282.214298

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

[13] Richard M Fujimoto. 1990. Performance of Time Warp Under Synthetic Workloads. In *Distributed Simulation (PADS '90)*, David Nicol (Ed.). Society for Computer Simulation International, San Diego, CA, USA, 23–28.

[14] Richard M Fujimoto. 1999. Exploiting Temporal Uncertainty in Parallel and Distributed Simulations. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS '99)*. IEEE Computer Society, Washington, DC, USA, 46–53. https://doi.org/10.1109/PADS.1999.766160

[15] Richard M Fujimoto, Rajive Bagrodia, Randal E Bryant, K Mani Chandy, David Jefferson, Jayadev Misra, David Nicol, and Brian Unger. 2017. Parallel Discrete Event Simulation: The Making of a Field. In *Proceedings of the 2017 Winter Simulation Conference*, Victor W K Chan, Andrea D'Ambrogio, Gregory Zacharewicz, Navonil Mustafee, Gabriel Wainer, and Ernest Page (Eds.). IEEE, Piscataway, NJ, USA, 262–291. https://doi.org/10.1109/WSC.2017.8247793

[16] Richard M Fujimoto and Maria Hybinette. 1997. Computing Global Virtual Time in Shared-Memory Multiprocessors. *ACM Transactions on Modeling and Computer Simulation* 7 (1997), 425–446. https://doi.org/10.1145/268403.268404

[17] David W Glazer and Carl Tropper. 1993. On Process Migration and Load Balancing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems* 4, 3 (March 1993), 318–327.

[18] Gerrit Großmann, Michael Backenköhler, and Verena Wolf. 2020. Importance of Interaction Structure and Stochasticity for Epidemic Spreading: A COVID-19 Case Study. In *Quantitative Evaluation of Systems*, Marco Gribaudo, David N Jansen, and Anne Remke (Eds.). Lecture Notes in Computer Science, Vol. 12289. Springer International Publishing, Cham, Switzerland, 211–229. https://doi.org/10.1007/978-3-030-59854-9_16

[19] Mauro Ianni, Romolo Marotta, Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2018. The Ultimate Share-Everything PDES System. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '18)*. ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/3200921.3200931

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

[21] David R Jefferson. 1990. Virtual time II: Storage Management in Conservative and Optimistic Systems. In *Proceedings of the 9th Symposium on Principles of Distributed Computing (PODC '90)*. ACM, New York, NY, USA, 75–89. https://doi.org/10.1145/93385.93403

[22] David R Jefferson and Peter D Barnes. 2022. Virtual Time III, Part 1: Unified Virtual Time Synchronization for Parallel Discrete Event Simulation. *ACM Transactions on Modeling and Computer Simulation* 32, 4 (Sept. 2022), 1–29. https://doi.org/10.1145/3505248

[23] Yi-Bing Lin and Edward D Lazowska. 1991. Processor Scheduling for Time Warp Parallel Simulation. In *Advances in Parallel and Distributed Simulation (PADS '91)*, David Nicol, Richard M Fujimoto, and Vijay Madisetti (Eds.). Society for Computer Simulation, San Diego, CA, USA, 11–14.

[24] Boris D Lubachevsky. 1989. Efficient Distributed Event-Driven Simulations of Multiple-loop Networks. *Commun. ACM* 32, 1 (Jan. 1989), 111–123. https://doi.org/10.1145/63238.63247

[25] Charles M Macal. 2010. To Agent-based Simulation from System Dynamics. In *Proceedings of the 2010 Winter Simulation Conference*, Björn Johansson, Sanjay Jain, and Jairo Montoya-Torres (Eds.). IEEE, Piscataway, NJ, USA, 371–382. https://doi.org/10.1109/WSC.2010.5679148

[26] Yukinori Matsumoto and Kazuo Taki. 1992. *Adaptive Time-Ceiling for Efficient Parallel Discrete Event Simulation*. Technical Report TR-0798. Institute for New Generation Computer Technology.

[27] Friedemann Mattern. 1993. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *J. Parallel and Distrib. Comput.* 18 (1993), 423–434. https://doi.org/10.1006/jpdc.1993.1075

[28] Horst Mehl. 1991. Speed-up of Conservative Distributed Discrete Event Simulation Methods by Speculative Computing. In *Proceedings of the Multiconference on Advances in Paralleland Distributed Simulation (PADS '91)*, Vijay Krishna Madisetti, David Nicol, and Richard M Fujimoto (Eds.). Society for Computer Simulation, San Diego, CA, USA, 163–166.

[29] David M Nicol. 1993. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. *J. ACM* 40, 2 (April 1993), 304–333. https://doi.org/10.1145/151261.151266

[30] David M Nicol and Xiaowen Liu. 1997. The Dark Side of Risk (What your Mother Never Told you About Time Warp). In *Proceedings of the 11th Workshop on Parallel and distributed simulation (PADS '97)*. IEEE Computer Society, Washington, DC, USA, 188–195. https://doi.org/10.1145/268826.268920

[31] Avinash C Palaniswamy and Philip A Wilsey. 1993. Adaptive Bounded Time Windows in an Optimistically Synchronized Simulator. In *Proceedings of the Third Great Lakes Symposium on VLSI (VLSI '93)*. IEEE Computer Society, Washington, DC, USA, 114–118. https://doi.org/10.1109/GLSV.1993.224467

[32] Avinash C Palaniswamy and Philip A Wilsey. 1993. An Analytical Comparison of Periodic Checkpointing and Incremental State Saving. In *Proceedings of the 7th workshop on Parallel and Distributed Simulation (PADS '93)*. ACM Press, New York, New York, USA, 127–134. https://doi.org/10.1145/158459.158475

[33] Avinash C Palaniswamy and Philip A Wilsey. 1994. Scheduling Time Warp Processes Using Adaptive Control Techniques. In *Proceedings of the 2004 Winter Simulation Conference*, Jeffrey D Tew, Mani S Manivannan, Deborah A Sadowski, and Andrew F Seila (Eds.). IEEE, Piscataway, NJ, USA, 731–738. https://doi.org/10.1109/WSC.1994.717422

[34] Alessandro Pellegrini and Francesco Quaglia. 2014. Wait-free Global Virtual Time Computation in Shared Memory Time Warp Systems. In *Proceedings of the 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '14)*. IEEE, Piscataway, NJ, USA, 9–16. https://doi.org/10.1109/SBAC-PAD.2014.38

[35] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2009. Di-DyMeLoR: Logging only Dirty Chunks for Efficient Management of Dynamic Memory Based Optimistic Simulation Objects. In *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS '09)*. IEEE, Piscataway, NJ, USA, 45–53. https://doi.org/10.1109/PADS.2009.24

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

[37] Andrea Piccione. 2022. Comparing Different Event Set Management Strategies in Speculative PDES. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '22)*. ACM, New York, NY, USA, 55–56. https://doi.org/10.1145/3518997.3534993

[38] Dhananjai M Rao, Narayanan V Thondugulam, Radharamanan Radhakrishnan, and Philip A Wilsey. 1998. Unsynchronized Parallel Discrete Event Simulation. In *Proceedings of the 1998 Winter Simulation Conference*, Deborah J Medeiros, Edward F Watson, John S Carson, and Mani S Manivannan (Eds.). WSC '98, Vol. 2.

IEEE, Piscataway, NJ, USA, 1563–1570. https://doi.org/10.1109/WSC.1998.746030

[39] Peter L Reiher, Frederick Wieland, and David Jefferson. 1989. Limitation of Optimism in the Time Warp Operating System. In *Proceedings of the 21st Winter Simulation Conference*, Edward A MacNair, Kenneth J Musselman, and Philip Heidelberger (Eds.). ACM, New York, NY, USA, 765–770. https://doi.org/10.1145/76738.76834

[40] Paul F Reynolds. 1988. A Spectrum of Options for Parallel Simulation. In *Proceedings of the 20th Winter Simulation Conference*, Michael A Abrams, Peter L Haigh, and John C Comfort (Eds.). ACM, New York, NY, USA, 325–332. https://doi.org/10.1109/WSC.1988.716181

[41] Paul F Reynolds, Chrisopher F Weight, and J Robert Fidler, II. 1989. Comparative Analyses Of Parallel Simulation Protocols. In *1989 Winter Simulation Conference Proceedings*, Edward A MacNair, Kenneth J Musselman, and Philip Heidelberg (Eds.). IEEE, Piscataway, NJ, USA, 671–679. https://doi.org/10.1109/WSC.1989.718741

[42] Tapas K Som and Robert G Sargent. 2000. Model Structure and Load Balancing in Optimistic Parallel Discrete Event Simulation. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS '00)*. IEEE, Piscataway, NJ, USA, 147–154. https://doi.org/10.1109/PADS.2000.847158

[43] Sudhir Srinivasan and Paul F Reynolds. 1998. Elastic Time. *ACM Transactions on Modeling and Computer Simulation* 8, 2 (April 1998), 103–139. https://doi.org/10.1145/280265.280267

[44] Jeffrey S Steinman. 1991. SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation. In *Advances in Parallel and Distributed Simulation (PADS '91)*, Vijay K Madisetti, David Nicol, and Richard M Fujimoto (Eds.). Society for Computer Simulation, San Diego, CA, USA, 1111–1115.

[45] Jeffrey S Steinman. 1993. Breathing Time Warp. *Simuletter* 23, 1 (July 1993), 109–118. https://doi.org/10.1145/174134.158473

[46] Tommaso Tocci, Alessandro Pellegrini, Francesco Quaglia, Josep Casanovas-Garcia, and Toyotaro Suzumura. 2017. ORCHESTRA: An Asynchronous Wait-free Distributed GVT Algorithm. In *Proceedings of the 21st International Symposium on Distributed Simulation and Real Time Applications (DS-RT '17)*. IEEE, Piscataway, NJ, USA, 1–8. https://doi.org/10.1109/DISTRA.2017.8167666

[47] Stephen J Turner and Ming Qiang Xu. 1991. Performance Evaluation of the Bounded Time Warp Algorithm. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS '92)*, Marc A Abrams and Paul F Reynolds, Jr (Eds.). Society for Computer Simulation, San Diego, CA, USA, 117–126.

[48] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. 2012. Load sharing for Optimistic Parallel Simulations on Multi Core Machines. *ACM SIGMETRICS Performance Evaluation Review* 40 (Dec. 2012), 2–11. https://doi.org/10.1145/2425248.2425250

[49] Minh Vu, Lisong Xu, Sebastian Elbaum, Wei Sun, and Kevin Qiao. 2022. Efficient Protocol Testing Under Temporal Uncertain Event Using Discrete-event Network Simulations. *ACM Transactions on Modeling and Computer Simulation* 32, 2 (March 2022), 1–30. https://doi.org/10.1145/3490028

[50] Philip A Wilsey, Avinash C Palaniswamy, and Sandeep Aji. 1994. Rollback Relaxation: A Technique for Reducing Rollback Costs in Optimistically Synchronized Parallel Simulators. In *Proceesings of the 1994 International Conference on Simulation and Hardware Description Languages (ICSHDL '94)*, David Rhodes and Philip A Wilsey (Eds.). Society for Computer Simulation, San Diego, CA, USA, 143–148.