Lightweight Time Warp – A Novel Protocol for Parallel Optimistic Simulation of Large-Scale DEVS and Cell-DEVS Models

Qi Liu, Gabriel Wainer
Department of Systems and Computer Engineering
Carleton University Centre on Visualization and Simulation (V-Sim)
Carleton University, Ottawa, Canada
{liuqi, gwainer}@sce.carleton.ca

Abstract

This paper proposes a novel Lightweight Time Warp (LTW) protocol for high-performance parallel optimistic simulation of large-scale DEVS and Cell-DEVS models. By exploiting the characteristics of the simulation process, the protocol is able to set free most logical processes (LPs) from the Time Warp mechanism, while the overall simulation still executes optimistically, driven by only a few full-fledged Time Warp LPs. The LTW protocol includes a rule-based event-scheduling mechanism using two types of event queues, an aggregated state-saving technique for optimal risk-free state management, and a new rollback algorithm that recovers lightweight LPs from causality errors without sending anti-messages. The impact on global control mechanisms such as GVT computation, fossil collection, and load migration is also discussed. The basic concepts of the protocol could also apply to a broad range of Time Warp systems under certain conditions and with appropriate control over the LPs.

1. Introduction

With the computing power and advanced software available today, Modeling and Simulation (M&S) has become a cost-effective tool for detailed analysis of a broad array of natural and artificial systems. Parallel and distributed simulation (PADS) is widely accepted as the technology of choice to speed up large-scale discrete-event simulation and to promote reusability and interoperability of simulation components. Originally introduced in [1], Jefferson's Time Warp protocol remains the most well-known optimistic synchronization algorithm that provides a solid foundation for many high-performance PADS systems. A Time Warp simulation is executed via several LPs interacting with each other by exchanging time-stamped event messages. The protocol consists of two distinct pieces that are sometime called the local control and global control mechanisms [2]. Rollback is used to recover LPs from causality errors upon the arrival of straggler or anti-messages with time stamps less than the LP's Local Virtual Time (LVT). Each LP maintains three data structures for this purpose: an *input queue* of recently arrived messages, an *output queue* of negative copies of recently sent messages, and a *state queue* of the LP's recent states. The historical events and states in these queues cannot be discarded until their time stamps have been surpassed by the Global Virtual Time (GVT). GVT computation and fossil collection are crucial components of the global control mechanism to reclaim memory resources and to commit I/O operations. Over the years, many algorithmic and data structure based optimizations have appeared in the literature to improve the efficiency of the original Time Warp protocol (e.g., [3]). The WARPED simulation kernel [7] is a configurable middleware that implements the Time Warp protocol and a variety of optimization algorithms.

The Discrete Event System Specification (DEVS) [8] is a general modeling framework for discrete-event systems. The P-DEVS formalism [9] eliminates the serialization constraint existed in the original DEVS definition, allowing increased parallelism to be obtained in PADS systems. The Cell-DEVS formalism [10] combines Cellular Automata [11] with DEVS theory to describe n-dimensional cell spaces as discrete event models, where each cell is represented as a DEVS basic model that can be delayed using explicit timing constructions. Both P-DEVS and Cell-DEVS formalisms are implemented in CD++ [12], an open-source M&S environment that supports standalone and parallel/distributed simulations on different platforms.

Although the Time Warp protocol has been discussed in a great number of studies, its applicability to simulating DEVS models is only rarely explored in the PADS literature (but see, e.g., [13]). Recently, we

developed a parallel optimistic simulator, called as PCD++ [15], for high-performance simulation of complex DEVS and Cell-DEVS models based on the WARPED simulation kernel. However, several technical challenges remain to be addressed to tackle the issues related to performance, scalability, and complexity of Time Warp based large-scale parallel simulation systems. Some of these issues are outlined as follows:

1. With a large number of LPs loaded on each available processor (or node), the memory resources can exhaust quickly due to the excessive amount of space used for saving past events and states. Hence, the simulator is forced to reclaim historical data with frequent GVT computation and fossil collection, an operation that itself is an important contributor to the overall operational overhead. Other existing algorithms such as *pruneback* [16], *cancelback* [17], and *artificial rollback* [18] attempt to recover from a memory stall only at the expense of additional computation and communication overhead. It is desired to have a protocol that can support large-scale optimistic simulation even when memory resources are tight, while at the same time reducing the overhead of GVT computation and fossil collection to the minimum, and do so only when absolutely necessary.

2. One potential performance hazard in large-scale optimistic simulation is that the cost of rollbacks increases dramatically simply because a massive number of LPs are involved in the rollback operation on each node. Prolonged rollbacks not only result in poor system performance, but also increase the probability of rollback echoes [2]. Therefore, it is imperative to fashion a new approach that can dramatically reduce the rollback cost without introducing much additional runtime overhead.

3. Different implementations of the event sets have been the focus of research for several years [19]. A primary motivation behind these efforts is to improve the efficiency of queue operations as the number of stored events increases in large-scale and fine-grained simulations. In addition to using advanced data structures, the simulation performance would also be improved if we could keep the event queues relatively short throughout the simulation, an alternative approach that warrants close examination.

4. Dynamic load balancing has been recognized as a critical factor in achieving optimal performance in large-scale PADS systems where the workload and communication patterns are in constant fluctuation [22]. Algorithms for dynamic load balancing usually rely on metrics whose values are valid only for a short period. Further, the impact of load migration should be minimized so that it does not severely interfere with the normal execution of the simulation system. This problem is especially severe in optimistic simulations since a potentially unbounded number of events and states associated with an LP must be transferred between processors. Only a few studies address specifically the issue of facilitating load migration in Time Warp systems. For example, Reiher and Jefferson proposed a mechanism to split an LP into phases to reduce the amount of data that must be migrated [24]. More recently, Li and Tropper devised a method that allows for reconstructing events from the differences between adjacent states so that only the state queue needs to be transferred [25]. However, this approach only works for systems with fine event granularity and small state size such as VLSI circuits. An agile load migration scheme is needed to reduce the overhead of dynamic load balancing in Time Warp systems.

In this paper, we address these issues by proposing a novel protocol, referred to as Lightweight Time Warp (LTW), for high-performance optimistic simulation of DEVS and Cell-DEVS models. The LTW protocol can effectively improve system performance in a variety of ways, including reduced memory consumption, lowered operational overhead for both local and global control mechanisms, more efficient queue operations, and facilitated load migration. We should stress that the LTW protocol can well be integrated with other widely accepted Time Warp optimizations to further improve the performance. Although our discussion is centered on parallel simulation in PCD++, the basic concepts of the LTW protocol could also apply to a broad range of Time Warp based PADS systems under certain conditions and with appropriate control over the LPs.

The remainder of the paper is organized as follows. Section 2 introduces the background on parallel simulation in PCD++. It also highlights the assumptions that underlie the LTW protocol. Section 3 proposes a rule-based event-scheduling scheme that utilizes two types of input queues to reduce memory consumption and to speed up the simulation. Section 4 describes an aggregated state-saving technique and an optimal risk-free state-saving strategy for efficient state management. Section 5 covers the rollback mechanism in the LTW protocol. The impact on the global control mechanisms is discussed in Section 6. Conclusion and future work are reported in Section 7.

2. Optimistic simulation in PCD++

A system is described in P-DEVS as a composition of behavioral (*atomic*) and structural (*coupled*) model components. The LPs are specialized into two categories: *simulators* and *coordinators*. A simulator is created for each atomic model to trigger the output and

state transition functions, while a coordinator is associated with a coupled model to keep track of the simulation time and to relay messages between its child simulators and the parent coordinator. To reduce communication overhead, PCD++ adopts a flattened structure consisting of four types of LPs: *Simulator*, *Flat Coordinator* (FC), *Node Coordinator* (NC), and *Root Coordinator*. Parallelism is achieved by partitioning the LPs onto multiple nodes. Figure 1 shows the PCD++ structure of the LPs on two nodes.

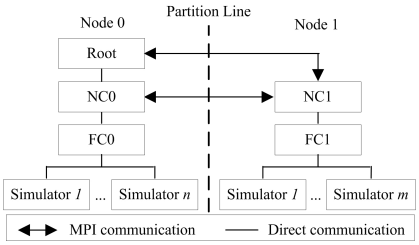

Figure 1. LP structure on two nodes

A single Root coordinator is created on node0 to start the simulation and to interact with the surrounding environment. Since the Root coordinator does not play a crucial role in the parallel simulation, we will not discuss it further for the sake of clarity. The simulation is governed by a set of NCs running asynchronously on different nodes in a decentralized manner. The NC acts as the local central controller on its hosting node and the endpoint of inter-node communication. It is the only LP responsible for determining the next simulation time on a node based on events from other remote NCs and the current states of the local LPs. The FC is in charge of routing messages between its child Simulators and the parent NC using the model coupling information. A Simulator executes DEVS functions defined in its associated atomic model upon the request of the FC. Note that on each node only one NC and FC are created, whereas many Simulators coexist in a typical large-scale simulation. Hence, a substantial reduction in the operational overhead at the Simulators would lead to a significant improvement in the overall system performance. Even though the LPs are grouped together, their LVT values may differ.

Messages exchanged between the LPs fall into two categories: *content messages* and *control messages.* The former includes external message (*x, t*) and output message (*y, t*) that encode the actual data transmitted between the model components, and the latter includes initialization message (*I, t*), collect message (*@, t*), internal message (**, t*), and done message (*D, t*) that are used to implement a high-level control flow in line with the P-DEVS formalism.

The message-passing organization has been analyzed in [15] using a high-level abstraction called Wall Clock Time Slice (WCTS). A WCTS at virtual time *t*, denoted as *WCTS-t*, stands for the execution of simultaneous events with time stamp *t* at all the LPs on a given node. As shown in Figure 2, the simulation process on each node is viewed as a sequence of WCTS's, each has a mandatory *transition phase* (T) and an optional *collect phase* (C). Only WCTS-0 has an additional *initialization phase* (I). Furthermore, a transition phase may contain multiple rounds of computation, denoted as $[R_0 \ldots R_n]$. During each round, state transitions are performed incrementally at the Simulators, incorporating additional (*x, t*)'s from the other model components. At the end of each WCTS, the NC calculates the next simulation time and sends out messages that will be executed by the local LPs at this new virtual time, initiating the next WCTS on the node. That is, the linking messages between two adjacent WCTS's have send time equal to the virtual time of the previous WCTS and receive time equal to that of the next. All other messages executed within a WCTS have the same send and receive time that is equal to the virtual time of the WCTS.

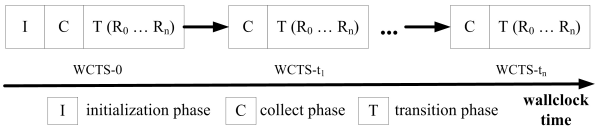

Figure 2. Simulation process represented in WCTS

Based on the LP structure and division of functionalities in PCD++, we summarize as follows the key characteristics of the simulation process, which also highlights the assumptions that underlie the LTW protocol.

1. The Simulators only communicate with their parent FC (i.e. no direct communication between the Simulators). Hence, the FC has the full knowledge of the timing of state changes at its child Simulators.

2. Rollbacks happened on a node begin either at the NC as a result of straggler or anti-messages arrived from other remote NCs, or at the FC in the case of messaging anomalies [15]. In both cases, rollbacks always propagate from the FC to its child Simulators. Hence, the FC has the information of when rollbacks will occur at the Simulators. Besides, a WCTS is an atomic computation unit for the FC and the Simulators during rollback operations [15].

3. The advance of simulation time on each node is controlled entirely by the NC. The FC and the Simulators do not send messages across WCTS boundaries.

These assumptions might seem a bit restrictive, but in practice many Time Warp based PADS systems can be converted, at least partially, into this model through

carefully choosing the level of event granularity and imposing an appropriate control over the LPs.

3. Rule-based dual-queue event scheduling

3.1. Introducing a volatile input queue

Under the Time Warp protocol, the input queue is *persistent*, in the sense that the events remain in the queue until being fossil collected when the GVT advances beyond their time stamps. However, keeping past events in the queue not only consumes lots of memory, but also increases the cost of queue operations. The LTW protocol solves this problem by introducing an additional *volatile* input queue that does not preserve processed events at all. Specifically, it is used to hold temporarily the simultaneous events exchanged between the FC and its child Simulators within each phase of a WCTS. On the contrary, the persistent input queue is used only by the NC and FC to contain the events sent between them. This arrangement is possible due to the simulation characteristics as we presented in the previous section. A key observation is that, during a rollback, the incorrect events previously exchanged between the FC and its child Simulators are essentially annihilated with each other. Therefore, it is safe to exclude these events from the persistent queue.

Figure 3 shows the message flow between four LPs: a NC, a FC, and two Simulators (S1 and S2). As we can see, events scheduled for the NC are still inserted into the persistent queue. However, events received by the FC are put into the persistent queue only if they are coming from the parent NC. In addition, all events exchanged between the FC and the Simulators are inserted into the volatile queue. Thus, these events are no longer controlled by the Time Warp mechanism. From the Time Warp perspective, the simulation process on a node only involves a small fraction of the total events executed by the LPs, as shown in Figure 4.

Comparing Figure 3 to Figure 4, we can see that the events executed by the FC and the Simulators within each phase of a WCTS can be considered as being collapsed into a single aggregated event. Note that the linking messages between adjacent WCTS's (e.g., $@_1$, x_{23}, $*_{24}$, and $@_{29}$) are still kept in the persistent input queue, which ensures that the simulation can resume forward execution successfully after rollbacks. In Figure 3, for instance, if a rollback with time t_2 occurs, then events x_{25}, $*_{26}$, D_{27}, D_{28}, and $@_{29}$ are cancelled and the simulation restarts with the unprocessed linking messages x_{23} and $*_{24}$ after the rollback.

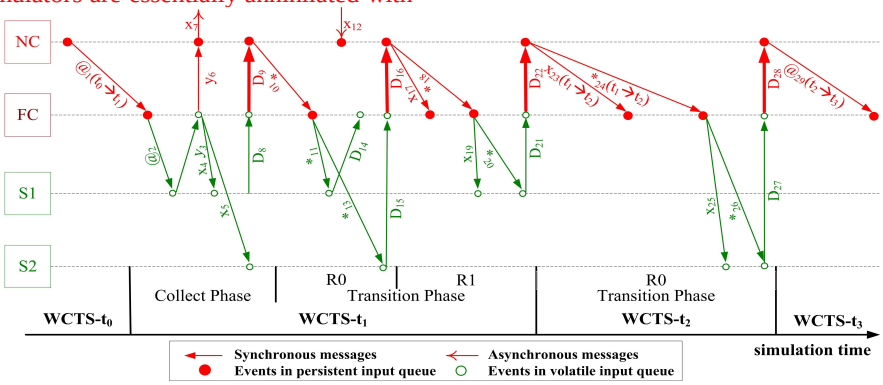

Figure 3. An example of message flow between the LPs on a node

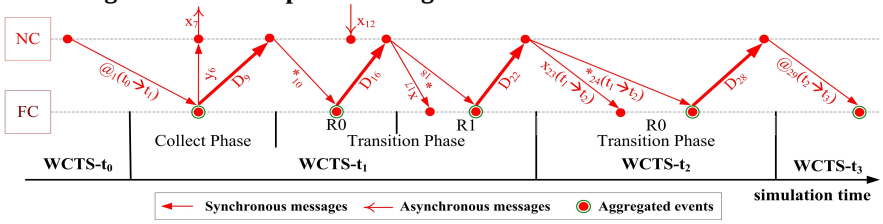

Figure 4. Message flow between the LPs from a Time Warp perspective

The volatile input queue has two appealing properties that allow us to reduce memory consumption and cost of queue operations significantly:

1. Events in the volatile queue are discarded and their memory reclaimed immediately after execution, greatly reducing the memory footprint of the system.

2. Events in the volatile queue always have the same time stamp. They are inserted into the queue as the simulation moves into each phase of a WCTS, and removed as the execution proceeds. At the end of each phase (i.e., when the FC sends a (*D*, *t*) to the NC), the queue becomes empty. This means that a simple FIFO

queue suffices, and queue operations can be performed efficiently in O(1) time. Events are simply removed from the head of the queue and added to the end. To enhance repeatability, the simultaneous events must be ordered in a repeatable fashion, such as by sorting on the ID of the receiving LP [2]. Thus, events may still need to be inserted in the middle of the volatile queue. However, the insertion operation is also accelerated in this case since the queue length remains relatively short throughout the simulation.

Consequently, the persistent input queue also becomes much shorter than under the original Time Warp protocol, allowing for more efficient queue operations as well. Moreover, for those events in the volatile queue, their counterpart anti-messages are no longer saved in the output queues of the sending LPs, further reducing the memory consumption and speeding up the forward execution of the simulation. Similarly, message annihilations are not required to cancel these events during rollbacks any more, minimizing the rollback overhead and enhancing the stability and performance of the system. The new rollback algorithm for the LTW protocol will be presented in Section 5. In addition, this approach also facilitates fossil collection due to the significant reduction in the number of past events and anti-messages stored in the persistent input and output queues, which, in turn, allows us to perform GVT computation and fossil collection more frequently without incurring an overwhelming computational expense, leading to even shorter queues. Impact of the LTW protocol on the global control mechanism will be analyzed in Section 6.

3.2. Rule-based event scheduling scheme

Although logically each LP has its own input queue, it is more convenient and efficient to create a single persistent input queue and a single volatile input queue that are shared by all the LPs mapped on a given node. With both queues at hand, we need to provide a proper scheduling policy that not only enforces a *Least-Time-Stamp-First* (LTSF) event execution on each node, but also helps improve execution efficiency and lower the possibility of performance degradation. The following discussion assumes that a scheduler is located on each node to determine the next event to be executed during each simulation cycle. Figure 5 illustrates the dual-queue event-scheduling scheme.

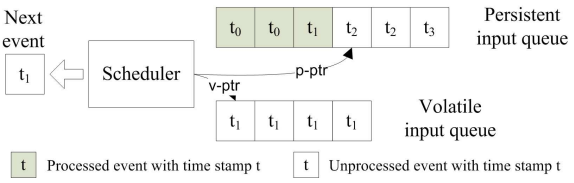

Figure 5. Dual-queue event scheduling

The persistent queue contains events sorted in LTSF order, including those unprocessed events and those have already been processed but not yet been fossil collected. On the other hand, the volatile queue only holds simultaneous events that have not yet been processed in the current phase of a WCTS. The scheduler maintains two pointers (`p-ptr` and `v-ptr`) to reference the next available events in the queues respectively. While `p-ptr` may need to be updated when the persistent queue is modified (event insertion and/or annihilation) to ensure that it always points to the first unprocessed event with the minimum time stamp, `v-ptr` is always pointing to the event at the front of the volatile queue. At each event selection point, the scheduler compares the two events referenced by the pointers based on a set of predefined rules and chooses one of them as the next event to be executed in the current simulation cycle. Different policies can be encoded in the scheduling rules, which essentially allow the scheduler to adjust the priorities of the input queues dynamically on an event-by-event basis.

During the execution of a WCTS, (*x*, *t*) from remote nodes may arrive and need to be processed by the NC (e.g., x_{12} arrives in R_0 of WCTS-t_1 in Figure 3). These messages will be flushed to the FC in the next round of the transition phase to be included in the state transitions at the Simulators (e.g., x_{17} is flushed to the FC in R_1 of WCTS-t_1). To avoid unnecessary rounds in the transition phase, the NC needs to execute the remote (*x*, *t*) immediately upon arrival. Likewise, the Simulators may send messages to receivers on remote nodes during the execution of a WCTS (e.g., y_3, y_6, and x_7). As these are potentially straggler messages at the receiving end, a delay in their delivery could postpone rollbacks at the destination, resulting in performance degradation. Bearing these factors in mind, we set the following scheduling rules, which grant a higher priority to the events in the persistent queue than those in the volatile queue if they have the same time stamp.

```
1.  when the scheduler is consulted to determine the next event
2.      if v-ptr = NULL (volatile queue is empty), then
3.          if p-ptr's timestamp > simulation stop time, then
4.              return NULL
5.          else
6.              next_event = p-ptr
7.              p-ptr = next unprocessed event in the persistent queue
8.              return next_event
9.          end if
10.     else
11.         if p-ptr's timestamp <= v-ptr's timestamp, then
12.             next_event = p-ptr
13.             p-ptr = next unprocessed event in the persistent queue
14.             return next_event
15.         else
16.             next_event = v-ptr
17.             remove the first event from the head of the volatile queue
18.             v-ptr = new head of the volatile queue
19.             return next_event
20.         end if
21.     end if
22. end when
```

Figure 6. Rule-based event scheduling algorithm

As shown in Figure 6, the next event is set to `NULL` if the volatile queue is empty and the next available event in the persistent queue has a time stamp greater than the simulation stop time (line 4). In this case, the simulation on this node is idle and the NC will reactivate the process later upon the arrival of (*x*, *t*) from the other nodes. When the volatile queue becomes empty at the end of each WCTS phase, the event pointed by `p-ptr` is selected (line 6), ensuring the NC can execute the (*D*, *t*) sent from the FC to initiate the next phase. If the volatile queue is not empty (i.e., the simulation is in the middle of a WCTS phase) and the next available event in the persistent queue has a time stamp that is equal to (during forward execution) or less than (after rollbacks) the time stamp of the events in the volatile queue, then the event pointed by `p-ptr` will be chosen (line 12). This guarantees that inter-node messages can be executed without being delayed and the simulation can resume forward execution immediately from the unprocessed linking messages after rollbacks. Otherwise, the scheduler selects the event pointed by `v-ptr` to execute in the current cycle (line 16), effectively enforcing an LSFT event execution. Note that an event selected from the volatile queue is removed (it will be deleted by the receiving LP after execution), whereas an event chosen from the persistent queue is simply marked as processed and the `p-ptr` is moved to the next available event afterward.

4. Aggregated state saving scheme

4.1. Introducing an aggregated state manager

In a Time Warp system, each LP has its own state manager that maintains a history of the LP's recent states in order to undo modifications to state variables during rollbacks. This approach allows for wide generality and straightforward implementation. However, it also suffers from several disadvantages. Firstly, the historical states are scattered among the individual LPs, prohibiting efficient batch operations from being applied to the state queues. For example, all the state queues must be queried individually during fossil collection, a costly operation that could otherwise be performed more efficiently in a more concentrated fashion. Secondly, state restorations at the LPs are triggered entirely by straggler and/or anti-messages, putting a tremendous burden on the underlying communication infrastructure. By exploiting the particularity of the simulation process, we introduce a new state-saving scheme that allows the Simulators to delegate the responsibility of state management to the FC. As a result, the Simulators are turned into truly lightweight LPs, totally isolated from the complex data structures required by the Time Warp mechanism.

At the heart of this state-saving scheme is an aggregated state manager created specifically for the FC. It not only takes care of the state queue for the FC itself, but also those used by the child Simulators. Conceptually, each Simulator still has its own state queue under the control of the aggregated state manager at the FC. It is perfectly possible, however, to employ other advanced data structures to achieve more efficient state queue operations. In addition, a Boolean "dirty bit" is associated with the state queue for each Simulator, as shown in Figure 7.

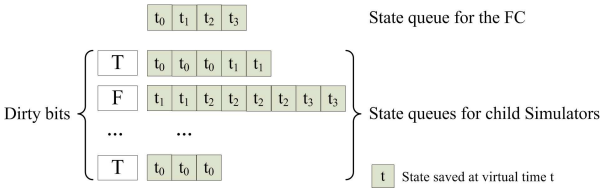

Figure 7. Aggregated state manager for the FC

As we mentioned in Section 2, a Simulator can change its state only if it executes an event coming from the FC. Hence, the FC knows the exact timing of when to save the state for a child Simulator. Nonetheless, the state should be saved after the processing of the event since the state variables in the destination Simulator will be modified during the event execution. Moreover, not all Simulators will be involved in the computation of a WCTS. Some of them may stay idle for an indefinite period. This is where the dirty bits have a role to play. After sending an event to a child Simulator, the FC simply instructs the aggregated state manager to set the corresponding dirty bit. The actual state-saving operation is carried out when the FC somehow detects that the events previously sent to the Simulators have already been processed, and is performed only for those Simulators with dirty bits set to true. Note that no dirty bit is associated with the state queue for the FC itself since

the FC is always involved in the computation of each WCTS.

4.2. Optimal risk-free state saving in PCD++

In [15], we proposed a Message Type-based State-Saving (MTSS) strategy that enables the LPs to save states only after executing certain types of events. Specifically, the NC and FC save state only after processing a (*D, t*), while the Simulators save state only after executing a (*, *t*). For instance, Simulator S1 will save two states in WCTS-t_1 after processing event $*_{11}$ and $*_{20}$ in Figure 3. Although the number of states saved in the simulation can be reduced significantly with the MTSS strategy, it is nevertheless suboptimal. Since a transition phase may have multiple rounds of computation, an LP could still save many states in each WCTS. For the purpose of state restoration, an optimal state-saving strategy should save only a single state for an active LP at the end of each distinct WCTS. The state-saving strategy we present here satisfies this condition of optimality. It is also risk-free in the sense that, unlike other infrequent state-saving techniques, no penalty is incurred as the result of saving fewer states.

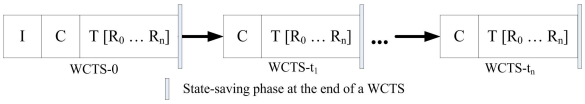

Figure 8. State-saving phase for each WCTS

As shown in Figure 8, the proposed strategy adds an extra state-saving phase to the end of each WCTS, where the NC determines the next simulation time during the execution of a (*D, t*) returned from the FC. If the simulation time advances to a new value, the NC will send linking messages to the FC to initiate the next WCTS on the node. However, instead of sending the messages immediately, the NC first instructs the FC to save states for the current WCTS. Note that at this moment all events belonging to the current WCTS have been processed by the FC and Simulators. Thus, the saved states will reflect the updated values of the state variables defined in the LPs. Only when the state-saving phase completes, can the NC send the linking messages to the FC to start the next WCTS for the new simulation time. The state of the NC itself is saved after processing the (*D, t*) from the FC, just like in a normal Time Warp execution.

```
1.  when the FC sends a (*, t) to a child Simulator S
2.      if S.dirty_bit = FALSE, then
3.          S.dirty_bit = TRUE
4.      end if
5.  end when

6.  when the NC requests a state-saving operation (at the end of each WCTS)
7.      save a copy of the FC's current state in the FC's state queue
8.      for each child Simulator S do
9.          if S.dirty_bit = TRUE, then
10.             save a copy of the Simulator's current state in its state queue
11.             S.dirty_bit = FALSE
12.         end if
13.     end for each
14. end when
```

Figure 9. State-saving algorithm

Figure 9 gives the state-saving algorithm for the aggregated state manager. The algorithm consists of two parts. When the simulation executes within a WCTS, the state manager simply sets the dirty bit for a child Simulator if the FC sends out a (*, *t*) (line 1 to 5). This is consistent with the MTSS strategy as a Simulator only needs to save states after executing internal messages. However, no state is actually saved until the simulation proceeds to the end of a WCTS and the NC is about to advance the local simulation time. At this point, the state manager first saves the FC's state in its state queue (line 7), and then saves states for all Simulators whose dirty bits have been set (line 10). After saving the states, the state manager resets the dirty bits back to false (line 11), ready to be used in the next WCTS.

Compared to the MTSS strategy, this new state-saving algorithm has two advantages. Firstly, the state manager only sets a Boolean flag for all but the last (*, *t*) sent to a Simulator within a WCTS, which can be performed much quicker than actually saving the states in the state queues, resulting in better system performance. Secondly, only one state is saved for each active Simulator in a WCTS regardless of how many rounds the transition phase may have, reducing memory consumption and the length of the state queues with accelerated queue operations.

5. Lightweight rollback mechanism

Up to now, our discussion has been centered on the forward execution of the simulation. However, one key issue remains to be addressed before the LTW protocol can be regarded as a viable mechanism. In the original Time Warp protocol, rollbacks are triggered by the arrival of straggler or anti-messages. With the events reclaimed immediately after execution in the volatile queue and the states delegated entirely to the FC, the Simulators cannot rely on this rollback triggering mechanism any more. Hence, we have to devise a new rollback algorithm for the LTW protocol.

Before delving into the details of the proposed rollback algorithm, let us summarize the LTW protocol presented so far.

1. The NC is the only full-fledged Time Warp LP on each node. All input events scheduled for the NC are stored in the persistent queue, and copies of messages sent to other LPs saved in the output queue. It has its own state manager in charge of state-saving operations with the NC's state queue. In other words, the NC executes as usual based on the standard Time Warp mechanism. Hence, the NC can be left out in our discussion of the new rollback algorithm.

2. The FC, however, becomes a mixed-mode LP serving as an interface between full-fledged and lightweight LPs. The input events of the FC are split between the persistent and volatile queues, and the anti-messages are saved in the output queue only for those events sent to the NC. While the FC needs to keep only a small fraction of the historical events in the LTW protocol, it assumes greater responsibility for state management on behalf of the child Simulators using the aggregated state manager. Since no anti-messages will be sent to the Simulators during rollbacks (these anti-messages are no longer saved in the FC's output queue), the challenge we are facing now is how to roll back the Simulators properly without using anti-messages.

3. The Simulators are turned into truly lightweight LPs, free from the burdens of maintaining historical data in their input, output, and state queues. They are neither expected nor allowed to carry out rollbacks on their own in LTW protocol, and thus can be excluded from the proposed rollback mechanism as well.

As we outlined in Section 2, rollbacks happened on a node always propagate from the FC to the child Simulators. Thus, the FC has the knowledge of when the rollbacks should occur at the child Simulators. Moreover, the incorrect input events previously executed by the Simulators do not need to be undone during rollbacks because they have already been deleted from the volatile queue during forward execution. This is one of the most elegant features of the LTW protocol since it can save a great amount of CPU time that would otherwise be wasted on matching the message and anti-message pairs in the input queues and annihilating a potentially large number of incorrect events. The result is an accelerated rollback process that could lead to a significant improvement in the overall system performance. The rollback of the FC itself is still triggered by straggler and/or anti-messages from the NC based on the standard Time Warp mechanism. The crux of the rollback algorithm thus lies in restoring the states of the Simulators to those that have been saved at the end of the last WCTS with virtual time strictly less than the current rollback time.

One difficulty is that the Simulators execute asynchronously and thus may not have the same LVT. During rollbacks, only the states of the Simulators that have been involved in the incorrect computation need to be restored. For example, if the current rollback time is 100, the state of a Simulator that has stayed idle since virtual time 80 should not be restored. As the state of the Simulator is not modified at or after the rollback time, it remains valid after the current rollback. To solve this problem, we introduce a simple bookkeeping procedure to the FC in order to keep track of the *latest state change time* (LCT) for each child Simulator. An array is created at the FC to record the latest virtual times when the states of the child Simulators are modified.

```
1.  when the simulation is started
2.      create a LCT vector of size N at the FC (N is the number of child Simulators)
3.      for i = 1 to N do
4.          LCT(i) = ZERO
5.      end for
6.  end when

7.  when the FC sends a (*, t) to a child Simulator i
8.      if LCT(i) != t, then
9.          LCT(i) = t
10.     end if
11. end when

12. when the FC receives a straggler/anti-message with time T (rollback time = T)
13.     rollback the FC using the standard Time Warp mechanism
14.     call scheduler.rollback_volatile_queue(T)
15.     for each child Simulator i do
16.         if LCT(i) >= T, then
17.             find the last state (S_last) saved before T in the Simulator's state queue
18.             restore the Simulator's state to S_last
19.             delete the states saved after S_last from the Simulator's state queue
20.             LCT(i) = S_last.LVT
21.         end if
22.     end for each
23. end when
```

Figure 10. Rollback algorithm for the FC

The rollback algorithm in Figure 10 focuses on the activities of the FC. The LCT array is created at the beginning of the simulation during the initialization phase. Initially, each LCT value is set to virtual time zero (line 4). Whenever the FC sends a (*, *t*) to a Simulator, the entry is updated to reflect the current LCT value for that Simulator (line 9). When a rollback occurs at virtual time T, the FC first takes all the necessary actions required by the Time Warp protocol to roll back its own speculative interactions with the NC (line 13). Then, the FC instructs the scheduler to roll back the events in the volatile input queue (line 14), which will be presented shortly. Finally, the FC restores the states for the child Simulators if necessary.

State restoration for a Simulator is performed only if the corresponding LCT value is greater than or equal to the rollback time (line 16), which means that the Simulator has participated in the incorrect computation and thus its state must be restored. The state restoration actions are carried out by the aggregated state manager in a similar fashion as the standard Time Warp mechanism (line 17 to 19). After the restoration,

however, an additional step is performed to update the LCT value to the LVT of the restored state (line 20). In this way, the FC can restore the states of the Simulators accurately during future rollbacks.

```
1. when rollback_volatile_queue(T) is invoked
2.       if v-ptr != NULL (volatile input queue is not empty), then
3.             if v-ptr's timestamp >= T, then
4.                   delete all events in the volatile input queue
5.             end if
6.       end if
7. end when
```

Figure 11. Rollback algorithm for the scheduler

As shown in Figure 11, the rollback algorithm for the scheduler is rather simple. The scheduler checks the status of the volatile input queue. If the queue contains events with a time stamp greater than or equal to the rollback time (i.e., these events have been scheduled but have not yet been executed in an incorrect WCTS), then the scheduler performs a batch operation to empty the volatile queue and reclaims the memory resources.

As we can see, rollbacks can be performed more efficiently in the LTW protocol than in the standard Time Warp mechanism due to, for the most part, the significant reduction of message annihilations in the persistent input queue. Moreover, all the Simulators can be rolled back without sending even a single anti-message no matter how many of them coexist on a node in large-scale simulations, dramatically reducing the communication overhead. The accelerated rollback process, in turn, decreases the likelihood of rollback echoes, enhancing the system performance and stability.

6. Impact on global control mechanisms

Although the LTW protocol can significantly reduce memory consumption and thus the frequency of GVT computation and fossil collection, these global control mechanisms still play a vital role in the parallel simulation. In this section, we briefly analyze the impact of the LTW protocol on the global control mechanisms.

Various GVT computation algorithms have been proposed in the PADS literature (e.g., [26]). In general, these algorithms require the GVT manager on each node to calculate its own local GVT estimation, based on which a new GVT value is computed and then broadcast to all the nodes in the system. Hence, the cost of GVT computation consists of two parts: the computational overhead for calculating local GVT estimations, and the communication overhead for collecting and broadcasting new GVT updates. The LTW protocol can mitigate the former overhead because it reduces the number of LPs that must be queried during the local GVT estimation on each node. However, it does not help reduce the communication between the GVT managers. Given that the communication overhead constitutes the major cost of GVT computation, the LTW protocol only has a minor effect on the operational efficiency.

On the other hand, the LTW protocol greatly improves the fossil collection process. Firstly, the persistent input queue has been shortened significantly after the introduction of the volatile input queue, and most of the anti-messages are eliminated from the output queues under the protocol, resulting in decreased overhead for reclamation of past events and anti-messages during fossil collection. Secondly, the aggregated state manager now controls most of the historical states in a more centralized manner, making it possible to perform batch operations with improved efficiency.

Dynamic load balancing is another type of global control that is crucial to achieving optimal simulation performance. In PCD++, model partitioning occurs at the atomic model level. Hence, it is the lightweight Simulators that will be moved around to achieve dynamic load balancing at runtime. Algorithms for dynamic load balancing usually rely on sensitive metric values that are valid only for a short period. Once a decision has been made, one or more LPs need to be migrated to a different node swiftly before the metric values become stale. Furthermore, the impact of load migration on the underlying communication infrastructure should be minimized so that it does not severely interfere with the normal execution of the simulation system. Under the LTW protocol, the appropriate decision points for load migration would be at the end of each WCTS when all the events in the volatile queue have been executed and the states of the LPs have been saved. Therefore, the only data that need to be transferred during load migration are the state queues of the chosen Simulators, lowering the cost of load migration considerably and allowing for more efficient dynamic load balancing to be realized in large-scale DEVS-based simulations.

7. Conclusion and future work

By taking advantage of the specific characteristics of the simulation process in PCD++, we proposed a novel protocol called as Lightweight Time Warp that is able to release most LPs from the Time Warp mechanism, while the overall simulation still executes optimistically, driven by only a few full-fledged Time Warp LPs. The LTW protocol can improve the system performance in various ways, including reduced memory footprint, lowered operational overhead for both local and global control mechanisms, more

efficient queue operations, and facilitated load migration. In addition, the LTW protocol can also be integrated with other widely accepted Time Warp optimizations to further improve system performance. Although our discussion is centered on parallel optimistic simulation of DEVS and Cell-DEVS models, the basic concepts could also apply to a wide range of Time Warp based PADS systems under certain conditions and with appropriate control over the LPs. We are currently implementing the LTW protocol and testing the performance quantitatively in PCD++.

10. References

[1] D. R. Jefferson, "Virtual time". *ACM Trans. Program. Lang. Syst. 7*(3), 1985, pp. 404-425.

[2] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*, Wiley-Interscience, 2000.

[3] J. Fleischmann, P. A. Wilsey, "Comparative analysis of periodic state saving techniques in time warp simulators". In *Proceedings of PADS'95*, Washington DC, 1995, pp. 50-58.

[4] J. S. Steinman, "Breathing time warp". *SIGSIM Simul. Dig. 23*(1), 1993, pp. 109-118.

[5] Y. B. Lin, E. D. Lazowska, "A study of time warp rollback mechanisms". *ACM Trans. Model. Comput. Simul. 1*(1), 1991, pp. 51-72.

[6] C. D. Carothers, K. S. Perumalla, R. M. Fujimoto, "Efficient optimistic parallel simulations using reverse computation". *ACM Trans. Model. Comput. Simul. 9*(3), 1999, pp. 224-253.

[7] R. Radhakrishnan, et al. "An object-oriented Time Warp simulation kernel". In *Proceedings of ISCOPE'98*, Vol. 1505, *LNCS*, Santa Fe, NM, 1998, pp. 13-23.

[8] B. P. Zeigler, H. Praehofer, T. G. Kim, *Theory of Modeling and Simulation*. Academic Press, London, 2000.

[9] A. C. Chow, B. P. Zeigler, "Parallel DEVS: A parallel, hierarchical, modular, modeling formalism". In *Proceedings of WSC'94*, San Diego, CA, 1994, pp. 716-722.

[10] G. Wainer, N. Giambiasi, "N-dimensional Cell-DEVS models ". *Discrete Event Dynamic Systems 12*(2), 2002, pp. 135-157.

[11] S. Wolfram, *A New Kind of Science*. Wolfram Media Inc., Champaign, 2002.

[12] G. Wainer, "CD++: A toolkit to develop DEVS models". *Software: Practice and Experience 32*(13), 2002, pp. 1261-1306.

[13] J. Nutaro, "Risk-free optimistic simulation of DEVS models". In *Proceedings of the Advanced Simulation Technologies Conference*, Arlington, VA, 2004.

[14] E. Glinsky, G. Wainer, "New parallel simulation techniques of DEVS and Cell-DEVS in CD++". In *Proceedings of IEEE ANSS'06*, Washington, DC, 2006, pp. 244-251.

[15] Q. Liu, G. Wainer, "Parallel environment for DEVS and Cell-DEVS models". *SIMULATION 83*(6), 2007, pp.449-471.

[16] B. R. Preiss, W. M. Loucks, "Memory management techniques for Time Warp on a distributed memory machine". *SIGSIM Simul. Dig. 25*(1), 1995, pp. 30-39.

[17] D. Jefferson, "Virtual Time II: Storage management in conservative and optimistic systems". In *Proceedings of PODC'90*, New York, NY, 1990, pp. 75-89.

[18] Y. B. Lin, B. R. Preiss, "Optimal memory management for Time Warp parallel simulation". *ACM Trans. Model. Comput. Simul. 1*(4), 1991, pp. 283-307.

[19] R. Brown, "Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem". *Commun. ACM 31*(10), 1988, pp. 1220-1227.

[20] R. Rönngren, R. Ayani, R. M. Fujimoto, S. R. Das "Efficient implementation of event sets in Time Warp". *SIGSIM Simul. Dig. 23*(1), 1993, pp. 101-108.

[21] S. Schof, "Efficient data structures for Time Warp simulation queues". *Journal of Systems Architecture 44*(6), 1998, pp. 497-517.

[22] P. Peschlow, T. Honecker, P. Martini, "A flexible dynamic partitioning algorithm for optimistic distributed simulation". In *Proceedings of PADS'07*, Washington, DC, 2007, pp. 219-228.

[23] M. Zhang, B. P. Zeigler, A. Boukerche, "Exploiting the concept of activity for dynamic reconfiguration of distributed simulation". In *Proceedings of IEEE DS-RT'07*, Chania, Crete Island, Greece, 2007, pp. 87-94.

[24] P. L. Reiher, D. Jefferson, "Dynamic load management in the Time Warp operating system". *Trans. Soc. Comput. Simul. Int. 7*(2), 1990, pp. 91-120.

[25] L. Li, C. Tropper, "Event reconstruction in Time Warp". In *Proceedings of PADS'04*, 2004, pp. 37-44.

[26] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation". *Journal of Parallel and Distributed Computing 18*(4), 1993, pp. 423-434.

[27] B. Kannikeswaran, et al. "Formal specification and verification of the pGVT algorithm". In *FME'96*, Vol. 1051, LNCS, 1996, pp. 405-424.