Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

VIRTUAL TIME III: UNIFICATION OF CONSERVATIVE AND OPTIMISTIC SYNCHRONIZATION IN PARALLEL DISCRETE EVENT SIMULATION

David R. Jefferson
Peter D. Barnes, Jr.

Lawrence Livermore National Laboratory
7000 East Avenue, L-561
Livermore CA 94550, USA

ABSTRACT

There has long been a divide in synchronization approaches for parallel discrete event simulation, between *conservative* methods requiring lookahead and *optimistic* methods requiring rollback. These are usually seen as dichotomous, so that a model writer must make an early, static design decision between them. An optimistic simulator does not need lookahead information but is unable to take advantage of it even if it were available, whereas a conservative simulator may perform poorly or even deadlock without good lookahead information. Here we introduce *unified virtual time* (UVT) synchronization which provides the advantages of both conservative and optimistic synchronization dynamically for all models. Conservative synchronization becomes an accelerator for optimistic synchronization. When lookahead information is available the simulation will execute conservatively. Otherwise it will execute optimistically. In this paper we present UVT, argue for its correctness, and show adaptations of Time Warp, YAWNS, and Null Messages which cooperatively synchronize a single simulation.

1 INTRODUCTION

Parallel discrete event simulation (PDES) is a widely used approach to modeling complex systems, especially those without governing differential equations. To construct a PDES model one decomposes the system into logical processes (LPs), each with its own set of state variables and its own local virtual time (LVT). LPs interact by sending timestamped *event messages* to each other, which trigger execution of model component code (*event handlers*) in increasing timestamp order to update the component's state.

Since almost the beginning of research on PDES there have been two fundamental approaches to synchronization, conservative methods that require *lookahead* information but not rollback, and optimistic methods that require *rollback* but not lookahead information (Fujimoto 2000). A conservative simulator executes an event only when it has a nontrivial lower bound on the timestamps on all event messages that will arrive in the future. An optimistic simulator speculatively executes events, correcting out-of-order execution by rolling back and re-executing the events in correct timestamp order.

Conservative simulators typically are easier to understand and implement, and tend to have lower event overhead, but they generally require structural restrictions on models and extra effort by the model writer to provide good lookahead information to achieve good performance. Optimistic simulators, on the other hand, though more complex, can deliver good performance over a broader range of models.

The two approaches have often been viewed as incompatible: a model writer must make an early static design decision to adopt one or the other (Carothers 2010). Models that could benefit from conservative synchronization in some parts or at some times but need optimistic synchronization in others must nonetheless make a static global choice.

In this paper we argue that it is possible to have it both ways—to gain the advantages of both conservative and optimistic execution *dynamically* and *within a single simulation*. With our *unified virtual time* (UVT) theory, events may be executed conservatively whenever good lookahead information is available at an LP, but otherwise they will be executed optimistically. The decision is made on an event-by-event

basis in each LP. The default synchronization method is optimistic, but the simulator automatically switches to conservative whenever timely lookahead information arrives at an LP, and back to optimistic when it can't execute conservatively any further. UVT thus treats conservative synchronization as an *optional accelerator* for optimistic synchronization or, conversely, it treats optimistic synchronization as a *baseline default capability* whenever good lookahead data is not currently available.

The UVT synchronization mechanism is compatible with essentially any conservative lookahead calculation algorithm and any rollback algorithm, with suitable refactoring to fit in the UVT framework described below. We will illustrate this with a basic optimistic Time Warp algorithm (Jefferson 1982; Jefferson 1985) along with simple versions of both the Chandy-Misra-Bryant (CMB) Null Message protocol (Chandy 1979; Bryant 1977) and a YAWNS-like windowing algorithm (Nicol 1993).

We have not yet implemented a UVT simulator, so we have no performance data to offer. We plan to implement it in the near future however, and invite others to do the same. There are many improvements to the algorithms presented here that we do not have space to describe but that will be obvious to implementers. Much work remains before the practicality of this scheme is demonstrated on real models.

This paper is organized as follows: Section 1.1 describes prior and related work. Section 2 describes our overall architecture and assumptions. Section 3 describes the UVT algorithm in detail. Section 4 shows how to implement YAWNS synchronization in UVT. Section 5 shows how to implement CMB synchronization in UVT. Section 6 offers some assessment and indicates future directions.

1.1 Prior work

There is a long history of prior work aimed at combining conservative and optimistic synchronization. Fujimoto described how to federate conservative and optimistic simulations in the context of the High Level Architcture (Fujimoto 1998). However each federate was statically designated as one or the other; they were not allowed to dynamically switch back and forth as proposed here.

Lubachevsky, *et al* described a variant of his Bounded Lag algorithm in which a limited amount of optimism ("filtered") is permitted to soften the restrictions of an otherwise fairly synchronous, time window-based conservative algorithm (Lubachevsky 1989).

Chandy and Sherman (Chandy 1989) described an innovative way of thinking about the unification of conservative and optimistic simulation in which a simulator is situated in a space-time coordinate system and a space-time relaxation algorithm calculates the state of the system at every point (event) in space-time. Those ideas were greatly elaborated upon and formalized in (Bagrodia 1991). The UVT system described here would fall within the nondeterministic universe of executions that the space-time relaxation algorithm could emulate, though nothing like UVT was specifically described.

In (Marotta 2016) the authors describe their lock free Share-Everything platform, designed specifically for a shared memory environment, and thus of limited scale. Recent versions do permit some LPs to execute optimistically up to one event beyond the conservative execution limit.

The first study we are aware of to attempt similar unification of conservative and optimistic synchronization (Jha 1994), in which LPs can switch dynamically between conservative and optimistic at any time. Mode changes are made by explicit request of the model code, however, rather than transparently by the simulator, so the model code must provide the switching logic. Also, the switch requires a rollback and possibly blocking, whereas UVT adds no overhead.

In (Perumalla 2005) the author describes a refined micro-kernel architecture for the μsik simulator, which allows a mixture of conservative and optimistic synchronization. However, each LP in that system is statically designated as either conservative or optimistic, whereas in UVT the kernel decides within each LP, on an event-by-event basis, whether to execute the event conservatively or optimistically.

To our knowledge UVT is the first synchronization mechanism proposed in which the simulator (as opposed to the model) decides dynamically, solely on the basis of whether lookahead data is available at the moment, whether to execute an event conservatively or optimistically.

2 UVT SIMULATOR ARCHITECTURE

First we discuss our basic assumptions about the computing platform, then the semantics of a UVT simulator, and finally the underlying architecture of the UVT simulator.

2.1 Computing Platform

Nodes, processors, cores: We assume a distributed-memory system, with each LP executed by its own simulation process. We make no assumptions about architecture, cores, threads, or co-processors.

Communication: We assume asynchronous, one-way, reliable, arbitrary-delay message transmission from any simulation executive or LP to any other. Structured communication packages such as MPI may simplify some parts of the simulator, but they bring constraints that we do not wish to impose generally.

Order preservation: In general we do not assume that the underlying message communication system preserves message order between any two LPs. As noted below, however, specific synchronization algorithms, in particular CMB, do require this constraint.

2.2 Simulator semantics

Communication graph: In general we do not assume a static communication graph connecting the LPs, nor even that the set of LPs is static—it may grow or shrink dynamically. Any LP can send an event message to any other at any time. However, some conservative synchronization algorithms, notably CMB, do require a static graph; regions of models using CMB must obey that restriction, as shown in Section 5.

Message cancellation: Message cancellation may be aggressive or lazy, and implemented *via* anti-messages or some other mechanism.

Conservative bit: Each message `m` contains a `conservative` bit set by the simulator to `true` when it is sent by a conservatively-executed event, and thus cannot be cancelled, or `false` when sent by an optimistically-executed event, and is subject to possible cancellation. This flag says nothing, however, about whether `m` itself will be processed conservatively or optimistically by its receiving LP.

Zero-delay events: We allow zero-delay events, *i.e*. an event at virtual time `t` may send an event message to be received at the same virtual time `t`, as long as there are no zero-delay cycles.

Event ties: Multiple event messages can arrive for the exact same virtual time at the same LP, a situation known as a *tie*. Tie-handling must be semantically identical whether events are executed conservatively or optimistically. The most general tie-handling mechanism, *superposition*, processes tied event messages together as a set. An event function `E` is called only once to process the entire set, and thus has the signature `void E(EventMsgSet evSet)`. Iterating over the elements of `EventMsgSet` must be deterministic, repeatable, portable, and independent of the order in which the elements were inserted.

Superposition is straightforward when executing optimistically. Arrival of an event message (or anti-message) which ties with a previously-executed event causes the simulator to roll back and re-execute with the modified tie-set. In conservative synchronization no event at virtual time `t` can be executed until the simulator can determine that *all* messages with that timestamp have arrived, from *all* possible senders.

Reversible and Irreversible event-handling functions: Any event function `E(evSet)` is represented by two related and almost equivalent functions (described further in Section 3.2):

- E^I`(evSet)` The *irreversible* version of `E(evSet)`, used for conservative execution.
- E^R`(evSet)` The *reversible* version of `E(evSet)`, used for optimistic execution.

Ideally the model programmer should only write E^I`(evSet)`, while E^R`(evSet)` is automatically generated at compile time, for example with Backstroke (Schordan 2015), or at runtime, using *e.g*. full state saving.

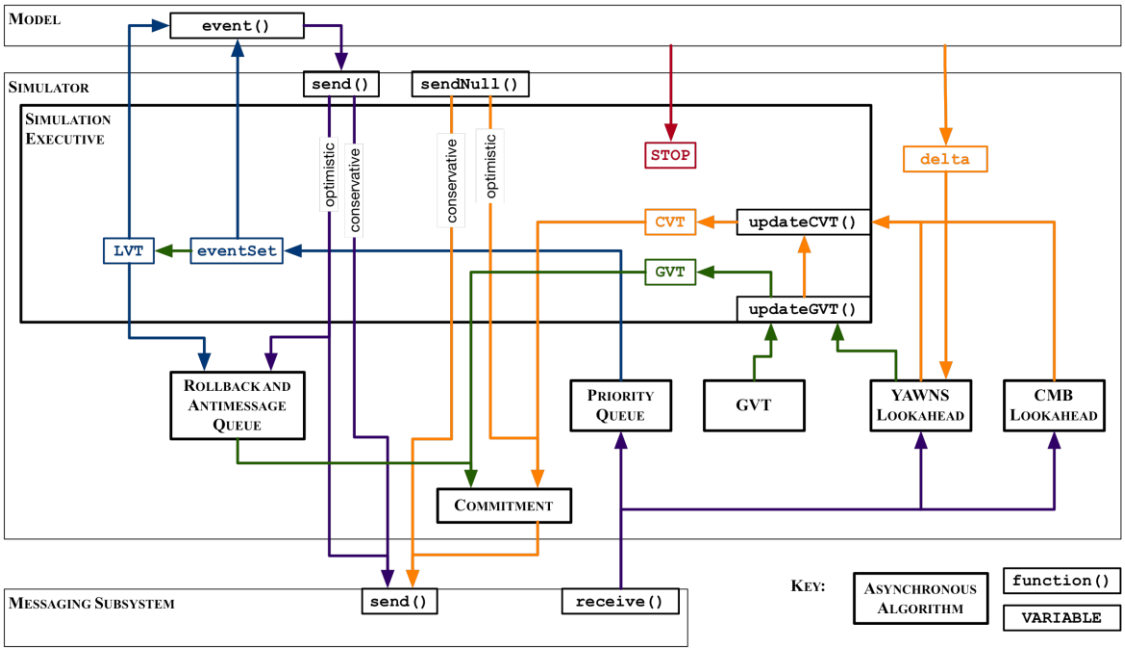

Figure 1. UVT simulator architecture.

2.3 Simulator architecture

The UVT simulator architecture is shown in Figure 1. The MODEL consists of event handler code and model state, and interacts with the simulator primarily by reading the current local virtual time (`LVT`) and by sending event and null messages. The SIMULATION EXECUTIVE is the main event loop. We separate several management algorithms from the main event processing loop and consider them to be asynchronous threads. These include: the MESSAGING SUBSYSTEM, PRIORITY QUEUE and ROLLBACK QUEUE management, `CVT` and GVT estimation algorithms, and COMMITMENT.

3 THE UNIFIED VIRTUAL TIME (UVT) SYNCHRONIZATION ALGORITHM

The basic idea of the UVT algorithm is that an LP executes events conservatively and irreversibly until it reaches a dynamically-calculated safe virtual time upper bound `CVT`, *conservative virtual time*. At this point a purely conservative simulator would pause to wait for updated lookahead information. Instead, the UVT simulator continues executing events beyond `CVT` optimistically. If an updated `CVT` value arrives at some later time some uncommitted events may be immediately committed, and if the new `CVT` is greater than `LVT` then UVT switches back to conservative execution. Switching back and forth between conservative and optimistic execution is handled at each LP on an event-by-event basis, transparently to the model code, without any model logic required to control it directly.

The `CVT` value at each LP is an upper bound on when it is safe for that LP to execute conservatively. The fundamental purpose of conservative synchronization algorithms in the UVT framework is to calculate good values for `CVT`, as often and with as little overhead as possible. If multiple independent CVT algorithms are used in the same simulation and each produces correct but different bounds for the same LP, then the simulator in effect maxes them, as shown below in `updateCVT()` in Section 3.6.

At any given moment some LPs may be executing conservatively, some optimistically, and some will be paused waiting for additional event messages, depending on the detailed dynamics of the model and the lookahead calculation. Just as in optimistic simulation, a UVT simulation is *internally nondeterministic* but *externally deterministic*, meaning that if a model is executed twice with identical configurations and inputs,

the set of events that execute conservatively or optimistically may differ, but the externally committed output will always be identical and repeatable.

We now present the UVT algorithm in more detail. UVT requires no changes to existing optimistic mechanisms, so we don't present them in detail. But in sections 4 and 5 we incorporate two conservative algorithms, variations on the YAWNS time windowing algorithm, and on the CMB Null Message algorithm. We combine these three (optimistic, YAWNS, CMB), but additional algorithms or variations could also be incorporated.

3.1 Virtual Time synchronization variables

The UVT simulator maintains at each LP three synchronization variables of type `vtime` (Virtual Time): `LVT`, `GVT`, and `CVT`. These are defined as follows:

`vtime LVT:` This is the traditional *local virtual time*, and is the virtual time of (1) the event currently executing in the LP or (2) if between events, the virtual time of the next event to execute, either conservatively or optimistically, or (3) if there are no more events to execute, then $+\infty$.

`vtime GVT:` This is the traditional estimated *global virtual time* used in all optimistic simulation algorithms. In each LP it is a lower bound on the (true, instantaneous, global) value TRUE_GVT, and on the virtual time of any LP state that will ever have to be restored in a rollback. LPs need not all have the same value of `GVT`, but all `GVT` values must be less than or equal to TRUE_GVT.

`vtime CVT:` This value, *conservative virtual time*, combines lookahead information from the CMB Null Message algorithm, the YAWNS algorithm and any other lookahead algorithms. It is the same as LBTS (lower bound on timestamp) used in other papers, and is a lower bound on the virtual time of any event message that will ever arrive at the LP in the future (including anti-messages, reverse (cancelback) messages, retraction messages, null messages, *etc*.) An LP can execute conservatively and events can be committed at virtual times up to but not including `CVT`. Note that while `GVT` is a *global* lower bound on rollback and future *event execution*, `CVT` is a *local* lower bound on future *message arrival*.

3.2 Conservative and optimistic event execution

We define the two modes of event execution as follows:

- *Optimistic execution:* For optimistic execution of event `E` the simulator calls the *reversible* event handler E^R`(evSet)`. Before and/or during its execution information is saved so that the LP state just before it is called can be reconstructed in the event of rollback. Event messages `m` sent by E^R`(evSet)` have the `m.conservative` flag set to `false`; corresponding anti-messages (or similar data structures) are saved so that event messages may be cancelled if necessary. Output operations and other commitment actions are delayed until commit time. Null messages sent by E^R`(evSet)` are held until commit time, as discussed in Section 5.
- *Conservative execution:* For conservative execution of event `E` the simulator calls the *irreversible* event handler E^I`(evSet)`. During E^I`(evSet)` no information needs to be saved to allow reconstruction of the prior state, and no provision needs to be made for cancelling event messages. Any message `m` generated by E^I`(evSet)` may be sent immediately and will have the `m.conservative` flag set to `true`.

An event `E` may be executed optimistically and then rolled back any number of times, before finally being cancelled, committed, or executed conservatively.

3.3 Invariants and the UVT principle

It is a consequence of the definitions in the previous sections that the UVT simulator must maintain the following critical invariants *at each LP*:

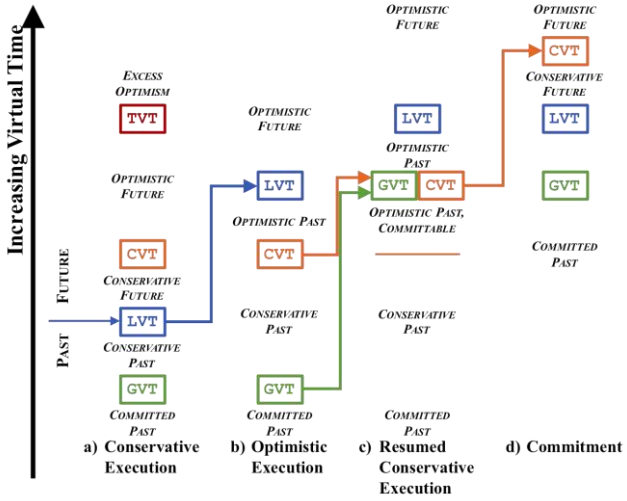

Figure 2: Relationships among LVT, GVT, CVT and conservative and optimistic execution.

Invariant 1: GVT ≤ LVT **Invariant 2:** GVT ≤ CVT

Invariant 1 follows directly from the definition of GVT as the lower bound on any future event.

Invariant 2 allows CVT to exceed GVT when there is lookahead available. Equality indicates that there is no lookahead information available for this LP at the moment.

The UVT Principle determines when an LP can execute an event conservatively or optimistically. It indicates that the simulator executes conservatively if and only if LVT < CVT, with the strict inequality required for correct tie-breaking during conservative execution.

UVT Principle: GVT ≤ LVT < CVT ⇒ LP executes conservatively
CVT ≤ LVT ⇒ LP executes optimistically

Combining the UVT Principle with the Invariants we conclude that when GVT == CVT, as it is by default, then the LP can only execute optimistically. If, however, CVT is frequently updated so LVT < CVT at all times, then the LP will always execute conservatively.

Each LP's instances of GVT and CVT are strictly monotonically increasing. LVT is monotonically increasing during conservative execution, but because of rollback it may not be monotonic during optimistic execution.

3.4 Commitment

In classic optimistic synchronization GVT is the commitment boundary. All queued actions earlier than GVT can be finalized: data and anti-messages for rollback can be discarded, I/O can be finalized, null messages transmitted, and runtime errors became fatal. In UVT this role of commitment boundary is played by CVT, not GVT (although of course they may be equal).

Even though GVT is not generally the commitment boundary, it still must be periodically calculated as a floor for CVT. If CVT is not updated by a lookahead algorithm to a value higher than GVT, then increases in GVT will force increases in CVT as a side effect to maintain Invariant 2, as shown in Algorithm 2 below.

3.5 The unified synchronization algorithm

Figure 2 illustrates how the UVT algorithm works. In panels a) through d) we show virtual time lines for a single LP. In panel a) we see that GVT < LVT < CVT. All events before GVT have been committed, and the

Algorithm 1: The main UVT event loop.

```
 1  vtime stoptime;                        // Stop time
 2  vtime LVT;                             // Local Virtual Time
 3  vtime CVT;                             // Local Committed Virtual Time
 4  PriorityQueue futMsgQ;                 // Queue of future event message sets
 5  PriorityQueue pastMsgQ;                // Queue of past event sets (for rollback)
 6  EventMsgSet evSet;                     // Next event message set
 7  while (GVT <= stoptime) do {           // Process events until stop time
 8    assert (GVT <= LVT);                 // Invariant 1
 9    assert (GVT <= CVT);                 // Invariant 2
10    if (!futMsgQ.haveEventSet) {         // No more event messages
11      LVT = +∞;
12      waitForEventMsg();                 // Wait for arrival of new event messages
13    }
14    evSet = futMsgQ.nextEventSet;        // Get the next event set and remove it
15    LVT = evSet.time;                    // Update LVT
16    if (LVT < CVT) {                     // UVT Principle
17      E^I(evSet);                       // Execute irreversibly, conservatively
18    } else {
19      E^R(evSet);                       // Execute reversibly, optimistically
20      pastMsgQ.enqueue(evSet);          // Save in pastMsgQ in case of rollback
21    }
22  }
```

simulator is executing events conservatively because `LVT < CVT`. Conservative events are committed as they execute, so no rollback data is saved, output and null messages are transmitted immediately, and any runtime error is immediately fatal.

In panel b) `LVT` has progressed past `CVT`, so events are now being executed optimistically. Rollback data is being saved, output and null messages are being queued, and runtime errors are tentative. Rollbacks can occur as necessary, but never to times earlier than `CVT`.

In panel c) `LVT` has advanced and a new higher value of `GVT` has been calculated, forcing up `CVT` to maintain Invariant 2. Events that were executed optimistically earlier than this new `CVT` value can now be committed. Since `LVT` > `CVT` the simulator continues executing optimistically.

In panel d), a new value of `CVT` has been calculated, now greater than `LVT`. Now all executed events can be committed, and the simulator can resume executing conservatively.

Algorithm 1 is a sketch of the Simulation Executive as it operates in the context of one LP.

Lines 1-6 define variables, including the future event priority queue and the rollback queue.

Line 7 starts the main event loop in an LP. The event loop terminates when `(GVT == +∞)` or `(GVT ≥ stoptime)` is detected.

Lines 8-9 check the two Invariants.

Lines 10-13 check for unprocessed messages in `futMsgQ`. If there are none, set `LVT = +∞` and wait for more event messages. If all LPs run out of event messages and no messages are in transit then `GVT == CVT == +∞` and the simulation will terminate normally (through actions in the commitment thread not shown here).

Line 14 removes the next unprocessed event message set with lowest virtual time from `eventMsgQ`.

Line 15 sets `LVT` to the virtual time of this event message set.

Lines 16-21 are the key lines in the simulator. They decide whether the event set `evSet` is executed conservatively and irreversibly, or optimistically and reversibly, based on the UVT Principle.

3.6 GVT and CVT updates

In principle at the start of a simulation `GVT` and `CVT` are both initialized to $-\infty$. This immediately satisfies Invariants 1 and 2. Therefore every simulation starts executing optimistically at every LP, by the UVT Principle, and can only execute conservatively once `CVT` has been updated to a value strictly greater than

Algorithm 2: Updating CVT and GVT.

```
1  void updateCVT(vtime newCVTestimate) {
2    CVT = max(CVT, newCVTestimate);    // This line must be done atomically
3    assert (GVT <= CVT);               // Invariant 2
4  }
5  void updateGVT(vtime newGVTestimate) {
6    GVT = max(GVT, newGVTestimate);    // This line must be done atomically
7    updateCVT(GVT);                    // Updating GVT may update CVT
8    assert (GVT <= LVT);               // Invariant 1
9  }
```

GVT. (In practice lookahead from the model may allow initializing CVT > GVT, so the simulation can start executing conservatively.) After initialization updated values of GVT and CVT are calculated from time to time. A simulation will execute correctly and make progress as long as no LP is forever starved of increases to GVT. The exact strategies for when and how to update GVT and CVT are unspecified here, and there can be considerable variation. In general they may be updated in separate threads asynchronously with event execution, though in some implementations it may be more convenient or provide better performance if they are updated synchronously. However CVT and GVT must be modified *only* through the functions shown in Algorithm 2.

Lines 1-4 update CVT. Line 2 guarantees that CVT is monotonically increasing, *i.e*. can never decrease, because this is the only line in the simulator that ever modifies CVT. Note that it is not an error for newCVTestimate to be smaller than the old CVT value. That can happen if two newCVTestimate values are calculated asynchronously and arrive at an LP out of order, either because they take different amounts of time to compute or suffer delivery delays. It is critical, however, that all messages with a timestamp less than or equal to newCVTestimate arrive before updateCVT(newCVTestimate) is called. If updateCVT() is never called, then GVT == CVT and only optimistic execution will be possible.

Line 3 checks that Invariant 2 is preserved.

Lines 5-9 update GVT. Line 6 guarantees that GVT is monotonically increasing. Again, it is not an error for newGVTestimate to be smaller than the old GVT value, for the same reasons as above. What matters for the correctness of the simulator is that every new value of GVT is in fact a true global lower bound on the virtual times of all current and future events for all LPs, *i.e*. GVT <= TRUE_GVT.

Line 7 forces a (possible) update to CVT whenever GVT is updated, to maintain Invariant 2.

Lines 8 checks that Invariant 1 is preserved.

It is important to recognize that GVT and CVT are distinct concepts and GVT is still necessary to guarantee the simulation's forward progress even though it is not always the commitment boundary. On the one hand GVT is just a value that gets maxed in to the calculation of CVT in Algorithm 2. On the other hand it is the *only* contributor to CVT that is strictly necessary for correctness of the UVT algorithm. If there is no lookahead in the model, or the lookahead calculation is just too slow, then periodic calculation of GVT is the only thing that will drive CVT forward and guarantee that the simulation still makes progress. As described in Section 1, we view conservative execution as an optional *accelerator* on top of a basically optimistic simulation engine.

4 INCORPORATING A YAWNS-TYPE WINDOWING ALGORITHM INTO UVT

Here we show how to implement a UVT-compatible conservative YAWNS algorithm, utilizing ideas from (Mikida 2016). To support this, the simulator needs an additional variable.

vtime delta: Static non-negative value (with default 0) set by model code at initialization time. This is the minimum virtual time delay to any future event from the current event, *i.e*. the minimum difference between the send time and receive time on any event message. It is an error to send an event message violating this constraint. (LP-local delta and dynamic adjustments to delta are possible, but beyond the scope of this paper.)

Algorithm 3: Asynchronous YAWNS thread.

```
 1  void AsynchYAWNS() {
 2    int epoch = 0;                    // Epoch counter
 3    vtime epochCVT = 0;               // CVT value at the beginning of the epoch
 4    int newCVT;                       // YAWNS-generated CVT estimate
 5    int msgSent[int];                 // Count of my messages sent, by epoch
 6    int msgRecv[int];                 // Count of my messages received, by epoch
 7    for (epoch = 0; ; ++epoch) {      // Complete each epoch in turn
 8      repeat
 9        int sends = msgSent[epoch];  // Global messages sent, reduce by sum
10        int recvs = msgRecv[epoch];  // Global messages recv'd, reduce by sum
11        vtime lvt = LVT;             // GVT estimate, reduce by min
12        AllReduce (sends, recvs, lvt);    // MPI-style blocking reduction
13      until (sends == recvs && epochCVT < lvt)
14      updateCVT(lvt + delta);        // New granted time window
15      epochCVT = lvt + delta;        // Update the epoch time
16    }
17  }
```

We take the classic YAWNS synchronization loop and run it asynchronously in successive epochs, as shown in Algorithm 3. The epochs are labeled by an index and the value of `CVT` at the start of the epoch. Within each epoch the Simulator Executive counts the number of messages sent and received (not shown in Algorithm 3). Then we proceed in the usual way: lines 8-13 perform a blocking MPI-style AllReduce on the YAWNS threads, summing the number of messages sent and received, and min-ing the next event time stamp, until all messages in the epoch have been received and all LPs have advanced past the epoch boundary `epochCVT`, which signals the end of the epoch. As in traditional YAWNS the new granted time is the minimum next event time plus the lookahead `delta` provided by the model, which we use to update `CVT` in line 14. Finally, we record the new epoch boundary in line 15.

5 INCORPORATING THE CMB NULL MESSAGE ALGORITHM CAST IN UVT FORM

In this section we incorporate our variant of the CMB Null Message synchronization algorithm (Chandy 1979) into UVT. CMB only applies when the model can provide *static* knowledge of the exact set of LPs that might send event messages to a receiver LP `r`. CMB also requires messages to be sent in non-decreasing timestamp order, and that order be preserved by the messaging layer. In our CMB version only some LPs need participate. We illustrate only a single "channel" between any particular sender and `r`.

In addition to sending event messages, the model can send a *null message* with timestamp `t` to an LP `r` using a special simulator call, `sendNullMsg(t, r)`. By sending a null message the sender guarantees that it will never send another event or null message to `r` with a timestamp less than `t`. Note, however, that `r` can still receive event messages with timestamps less than `t` from *other* senders.

We do not assume a sender knows if a destination LP is using CMB, which has two consequences. First, LPs potentially sending to CMB receivers must send messages in non-decreasing timestamp order; violating this restriction will cause a fatal error. Second, while it is safe to send null messages to non-CMB LPs they will just be discarded, at the cost of wasted time and bandwidth.

In our version of CMB null messages can be sent regardless of whether the sending LP is executing optimistically or conservatively. If the LP is executing conservatively null messages are transmitted normally, *i.e.* immediately. But since null messages are intended to communicate lookahead guarantees, during optimistic execution they must be queued at the sender until the sending event is committed. If an event is rolled back, any queued null messages it sent are discarded. Null messages from an optimistically executed event are thus treated essentially like output, transmitted only at commit time.

Having defined our version of CMB we now discuss the implementation.

Null bit: Our version of the CMB synchronization algorithm uses null messages, which we indicate with a `null` bit in the message header. `m.null` is `true` for a null message, and `false` for an ordinary event message. Null messages always have `m.conservative == true`.

Algorithm 4: Processing incoming messages incorporating CMB.

```
 1 void processIncomingMsg (vtime t, LPname src, msg m) {
 2   if (CMB && m.conservative) {        // CMB-specific processing
 3     int s = indexOf(src, sender);     // Indicate not found by returning -1
 4     if (s>=0) {                       // Sender src found among senders
 5       assert (CVT <= senderCVT[s]);   // Check CVT consistency
 6       assert (senderCVT[s] <= t);     // Check non-decreasing times from s
 7       senderCVT[s] = t;               // Update CVT for channel s
 8       updateCVT(min(senderCVT);       // Update CVT
 9     } else fatalerror(t, src, m, "Message arrived from unexpected sender");
10   } // if CMB
11   eventMsgQ.insert(t,m);              // Enqueue m
12 }
```

Boolean CMB: `true` if this LP is using the CMB synchronization, and `false` otherwise.

int senderCount: The number of LPs that can send event messages to this LP.

LPname sender[senderCount]: A complete list of all LPs that can send messages to this LP. It is an error for an event or null message to arrive from an LP not on this list.

vtime senderCVT[senderCount]: For each sender, `i`, to this LP `senderCVT[i]`, initialized to $-\infty$, contains a lower bound on the timestamps of all future messages that will ever arrive from `sender[i]`. *senderCVT*`[i]` can only monotonically increase. If it is ever set to $+\infty$, no further messages must ever come from `sender[i]`. The minimum of all elements in the *senderCVT* array is a lower bound on all future messages to this LP from any source, and thus can safely be used as a value for `CVT`.

Algorithm 4 shows how an arriving message `m`, either a null message or a regular event message, from sending LP `src` and with timestamp `t`, is processed in the simulator.

Lines 2-10 implement the CMB algorithm. Line 2 checks that this LP is CMB-capable and the message has valid lookahead information. `conservative` event messages provide valid lookahead information. (Recall that null messages are `conservative`). A non-`conservative` event message provides no useful lookahead information, since it might be rolled back.

Line 3 searches the list of permitted senders to this LP to find the message sender's index.

Line 4 checks that the sender is indeed a permitted sender. If it is not, line 9 raises a fatal error.

Line 5 tests that `CVT` is still consistent with the sender's last CVT value.

Line 6 checks that the stream of null and conservative event messages from `src` are indeed in non-decreasing virtual time order, as required. It is a fatal error if not.

Line 7 records `t` as the new lower bound on the timestamps of messages from LP `src`.

Line 8 is the key line in the algorithm. It updates `CVT` with the minimum of the `senderCVT` array, which is now a lower bound on all future messages from any permitted sender.

In line 11 the simulator finally queues the event message for eventual execution by the model.

6 EXTENSIONS AND CONCLUSION

In this paper we have demonstrated that conservative and optimistic PDES synchronization are not mutually exclusive, but can be unified harmoniously in a natural, scalable way, so that events will execute conservatively when there is lookahead information available that permits it, but optimistically otherwise. We truly can have the best of both worlds.

There is much more that can be done to extend and elaborate the UVT algorithm described in this paper. Other conservative lookahead calculation algorithms can be added to the two described here, and as long as they calculate correct lower bounds on future message arrival times and combine them into `CVT` using only the `updateCVT()` function, the simulation will synchronize correctly. Additional lookahead algorithms may capture lookahead information that YAWNS and CMB do not, so new `CVT` values they produce may be systematically higher for some models. Or new algorithms may produce lookahead data faster, or more frequently, or for LPs not covered by other algorithms. Of course each new lookahead algorithm has to

produce additional performance to cover the cost of its calculation, but within that constraint there is no limit to the number that can be combined into the UVT framework.

It is also possible to extend UVT to include other synchronization-like phenomena, for example *throttling* of optimistic execution. We can attach another virtual time parameter, `TVT`, to each LP to serve as the upper bound to *optimistic* execution. An LP executes conservatively up to `CVT`, then optimistically up to `TVT`, and then pauses until either a rollback or until `TVT` is increased. The simulator would then maintain extended Invariant relations and an extended UVT Principle for each LP as follows:

Invariant 1: `GVT ≤ LVT < TVT` **Invariant 2:** `GVT ≤ CVT < TVT`

UVT Principle: `GVT <= LVT < CVT` ⇒ LP executes conservatively
`CVT <= LVT < TVT` ⇒ LP executes optimistically

As is well known, individual synchronization algorithms have unique requirements for the kinds of input needed from a model, and assumptions/constraints on the simulation executive (particularly around message delivery). YAWNS requires model information about minimum send-to-receive simulation time `delay` on event messages; CMB requires static model information about the communication graph, and non-decreasing timestamps on messages, as well as in-order delivery from the messaging system. Many forms of rollback for optimistic simulation also require input from the model, in the form of reversing functions, identifiable LP state, *etc*. UVT does not relax these constraints. Rather it enables model writers to provide the information which they have, and let UVT execute the model as fast as possible. Whenever the constraints of all the cooperating synchronization algorithms are met, conservative or optimistic, we believe that the UVT framework can enable seamless combination of all of them.

UVT synchronization enables an optimized software engineering strategy for model developers. In the early stages, they should concentrate on getting the model logic correct under purely optimistic synchronization as the default. The early model code should be instrumented to measure space utilization and identify performance bottlenecks, but otherwise should concentrate on functionality, verification, and validation. Later, as performance becomes more of a priority and the sources of lookahead in the model's critical path are better understood, the developers can add logic to calculate lookahead data or improve it, with the goal of achieving additional speed through conservative execution.

ACKNOWLEDGEMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, as LLNL-CONF-733763. We thank anonymous referees for pointing us to papers describing related ideas that greatly improved our discussion of prior work.

REFERENCES

Bagrodia, R., K. M. Chandy, and W. T. Liao. 1991. "A Unifying Framework for Distributed Simulation". *ACM Transactions on Modeling and Computer Simulation*, 1(4): 348-385.

Bryant, R. E. 1977. "Simulation of Packet Communication Architecture Computer Systems". M.I.T. MS thesis, MIT/LCS/TR-188.

Carothers, C. D., and K. S. Perumalla. 2010. "On Deciding Between Conservative and Optimistic Approaches on Massively Parallel Platforms". In *Proceedings of the 2010 Winter Simulation Conference*, edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, 678-687. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Chandy, K. M., and J. Misra. 1979. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs". *IEEE Transactions on Software Engineering*, SE-5:440-452.

Chandy, K. M., and R. Sherman. 1989. "Space-Time and Simulation". *Distributed Simulation 1989*, The Society for Computer Simulation. University of Southern California Information Sciences Institute ISI Reprint Series #238.

Fujimoto, R. M. 1998. "Time Management in the High Level Architecture". *Simulation*,71(6):388-400.

Fujimoto, R. M. 2000. *Parallel and Distributed Simulation Systems*. New York: John Wiley & Sons, Wiley InterScience.

Jefferson, D. R., and H. Sowizral. 1982. "Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control". Rand Note N-1906-AF, The Rand Corporation.

Jefferson, D. R. 1985. "Virtual Time". *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404-425.

Jha, V., and R. Bagrodia. 1994. "A Unified Framework for Conservative and Optimistic Distributed Simulation". In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*, 12-19. New York: ACM SIGSIM Simulation Digest, 24(1):12-19.

Lubachevsky, B., A. Schwartz, and A. Weiss. 1989. "Rollback Sometimes Works … If Filtered (Abstract)". In *Proceedings of the 21st Winter Simulation Conference*., edited by E. A. MacNair, K. J. Musselman, P. Heidelberger, 630-639. New York: ACM.

Marotta, R., M. Ianni, A.Pellegrini, and F. Quaglia. 2016. "A Lock-Free O(1) Event Pool and Its Application to Share-Everything PDES Platforms". In *Proceedings of the IEEE/ACM Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, 53-68.

Mikida, E., L. V. Kale, E. Gonsiorowski, C. D. Carothers, P. D. Barnes, Jr, and D. R. Jefferson. 2016. "Towards PDES in a Message-Driven Paradigm: A Preliminary Case Study Using Charm++". In *Proceedings of PADS 2016*, 99-110. New York: ACM.

Nicol, D. M. 1993. "The Cost Of Conservative Synchronization In Parallel Discrete Event Simulations". *Journal of the Association for Computing Machinery,* 40(2): 304-333.

Perumalla, K. 2005. "μsik – A Micro-Kernel for Parallel/Distributed Simulation Systems". In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, 59-68. Washington, D.C.: IEEE Computer Society.

Schordan, Markus, D. Jefferson, P. Barnes, Jr., T. Oppelstrup, and D. Quinlan. 2015. "Reverse Code Generation for Parallel Discrete Event Simulation". In *Proceedings of 7th Conference on Reversible Computation*, edited by Jean Krivine and Jean-Bernard Stfani, 95-110. Switzerland: Springer International.

AUTHOR BIOGRAPHIES

DAVID JEFFERSON is a Visiting Scientist at Lawrence Livermore National Laboratory. He holds a Ph.D. in Computer Science from Carnegie Mellon University, and has worked in the field of parallel discrete event simulation (PDES) for decades. He is the co-inventor (with Henry Sowizral) of optimistic methods for PDES and was the architect of the first optimistic simulator, the Time Warp Operating System. He has also worked in various other fields, including distributed computation, synchronization, cybersecurity, public election security, and artificial life. His email address is drjefferson@llnl.gov.

PETER D. BARNES, JR. is a staff scientist at Lawrence Livermore National Laboratory. He holds a Ph.D. in Physics from the University of California, Berkeley. His research interests include the fundamentals of PDES, as well as adapting PDES to new application areas. He has served on the technical program committees of PADS, WinterSim and WNS3, for which he has also served as Program Committee Chair. He is a reviewer for TOMACS. He is the current "core" maintainer for the ns-3 simulator. His email address is pdbarnes@llnl.gov.