Proceedings of the 2013 Winter Simulation Conference
R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, eds.

REAL-TIME SCHEDULING OF LOGICAL PROCESSES FOR PARALLEL DISCRETE-EVENT SIMULATION

Jason Liu

School of Computing and Information Sciences
Florida International University
Miami, Florida 33199, USA

ABSTRACT

We tackle the problem of scheduling logical processes that can significantly affect the performance of running parallel simulation either in real time or proportional to real time. In particular, we present a comprehensive solution to dealing with the mixture of simulated and emulated events in a full-fledged conservatively synchronized parallel simulation kernel to improve efficiency and timeliness for processing the emulated events. We propose an event delivery mechanism for the parallel simulator to incorporate emulated events originated from the physical system. We augment the parallel simulation API to support emulation capabilities independent from a particular simulation domain. Preliminary experiments demonstrate the our simulator's real-time performance on high-performance computing platforms.

1 INTRODUCTION

Simulation is widely used for understanding the behavior of complex systems and interacting processes. There has been significant interest in making simulation capable of interacting with the physical systems and conducting either human-in-the-loop or machine-in-the-loop studies. For example, interactive simulation has been used for constructing Distributed Virtual Environments (DVEs) in the context of military training and online gaming. Interactive simulation has also been an essential property of the High Level Architecture (HLA), which also focuses on interoperability among simulators of various types. One important application is *on-line simulation*, where the simulator monitors the performance of the physical system in real time, conducts fast simulation experiments to evaluate alternative solutions, and then reconfigures the physical system for the best outcome. In this case, the simulator and the physical system form a control-feedback loop in order to improve the performance of the physical system. Another important application is *real-time simulation*, where the simulation is run in sync with the wall-clock time so that it can interact with the user or the physical system by exchanging real-time events. In this scenario, the simulator and the physical system together provide a closely integrated environment for testing components and applications.

Both on-line simulation and real-time simulation are special cases for *symbiotic simulation*, which is epitomized by a mutual beneficial relationship between the simulator and the physical system (Fujimoto et al. 2002). In either case, the simulator must be able to interact with the physical system with specific timing requirements. For on-line simulation, the simulator may need to run faster than real time so that the results can be generated promptly and used to steer the physical system. For real-time simulation, the exchange of events between the simulator and the physical system must be performed in real time at proper timing granularity dependent upon the specific applications.

Parallel discrete-event simulation (Fujimoto 1990) is a technique of running simulation concurrently on parallel platforms. In general, a simulation model is decomposed into a collection of logical processes (LPs), each containing a local event list with its own simulation clock. Synchronization protocols allow the LPs to coordinate with one another so that the events can be processed in non-decreasing timestamp at the corresponding LPs to ensure causality. Parallel simulation has been shown to achieve significant speedup over sequential simulation, and thus can enable large-scale simulation either faster than or in sync with the real time. In this paper, we focus on the problems of enabling parallel simulation to run either in real-time or in proportion to real time.

The traditional technique of running real-time simulation is to "pace" the simulation events according to the wall-clock time. That is, the simulator will suspend its execution if it finds that the timestamp of the next event on the event list is greater than the current wall-clock time (which is usually offset by the wall-clock time when the simulation starts). This technique is intuitive and reasonable in the context of sequential simulation, as it was first used for supporting network emulation in *ns-2* (Fall 1999). For parallel simulation, however, it creates a problem where multiple LPs may need to be paced simultaneously in real time. One way is simply have the simulator to dispatch the LPs in order of their simulation clocks, e.g., as in IP-TNE (Simmonds et al. 2000). When an LP is chosen to run, it first uses a conservative parallel synchronization protocol to determine the lower bound on timestamp (LBTS) of the incoming future events from other LPs, and then processes the local "safe" events—those with the timestamp smaller than LBTS—all in one batch. The problem is, since the simulator needs to process all safe events on one LP in real time before switching to other LPs, the events on the other LPs may turn out missing their real-time deadlines. To avoid this problem, the simulator can preempt the processing of events on an LP and switch to another LP with an earlier next event. One can achieve the same effect by merging all the LPs (and their event lists) located at the same processor. Both solutions have been used commonly by parallel real-time simulators today. Since, in both cases, the simulator needs to pin down all simulation events to real time, it basically sequentializes all simulation events on the processor and thus may destroy the potential parallel opportunity provided by the LP modeling paradigm.

Overall, we observe that a systematic treatment of parallel real-time simulation is missing in literature. More specifically, important questions still remain to be addressed, which include:

1. How to differentiate *emulated events*, which need to be pinned down to real time, from *simulated events*, which do not require real-time processing? Subsequently, how to maintain the causality between the events determined by the natural timestamp ordering of the events?
2. How to schedule LPs to ensure: (1) *timeliness*, which determines the capability of observing the real-time deadlines of the emulated events, and (2) *responsiveness*, perceived as the lag between the occurrence of the events in the physical system and their processing in the simulator?
3. How to design a generic application programming interface (API) to conveniently express the models consisting of both simulated and real-time components?

In this paper, we present a comprehensive solution to dealing with the mixture of simulated and emulated events in a full-fledged conservatively synchronized parallel simulation system. In particular, we make a careful distinction between processing simulated and emulated events in order to achieve better efficiency and improve the timeliness of the emulated events. The event delivery mechanism of the parallel simulator is designed to easily and efficiently incorporate emulated events from the physical system. Furthermore, we implement the real-time mechanism in our high-performance parallel simulator (MiniSSF 2013). We augment the parallel simulation API with generic functions to support the emulation capabilities independent from any specific simulation domain. The new API allows coherent real-time execution of the emulated events in simulation and supports flexible interaction between the simulator and the physical system.

2 BACKGROUND AND RELATED WORK

In this section, we discuss the background and identify important related work in areas of parallel discrete-event simulation, interactive simulation, as well as network emulation and simulation.

2.1 Parallel Discrete-Event Simulation

Time management is the central theme of parallel discrete-event simulation (also called parallel simulation, in short) (Fujimoto 1990). In general, parallel simulation adopts spatial decomposition: a simulation model is expressed as a set of interconnected logical processes (LPs) which can be assigned to different processors for parallel processing. By exploiting the concurrency between the LPs, parallel simulation is able to accelerate simulation, to run large-scale models, or both. A critical issue of parallel simulation is to maintain the *local causality constraint*, which requires events at each LP be processed in non-decreasing timestamp order. Parallel simulation can be classified into two broad regimes—conservative and optimistic synchronization—based on the differences in the methods used to enforce causality.

Conservative synchronization, such as the CMB protocol (Chandy and Misra 1979), strictly prohibits out-of-order event processing: an LP can process its local events only when it is guaranteed that other LPs will not later send events to it with smaller timestamps. Studies suggest that the performance of conservative synchronization protocols is heavily influenced by the *lookahead*, which can be regarded as a model-dependent lower bound for the simulation clocks at the LPs to drift away from one another. Lookahead essentially underlines the ability of the simulation model at predicting future events. For conservative synchronization, it is important to extrapolate the lookahead explicitly from the simulation model in order to achieve good parallel performance. Optimistic synchronization, such as Time Warp (Jefferson 1985), allows events to be processed possibly out of order. Once a causality error is detected, the simulator rolls back the execution to a state before the time of the event to recover from the erroneous computation. To achieve good efficiency, time warp needs to address important problems, such as state saving, rollback, global virtual time (GVT) computation, and memory management.

Theoretically, both conservative and optimistic synchronization can support real-time simulation. However, we observe that, for optimistic synchronization, since real-time events may directly deal with the physical system causing irrevocable operations that cannot be rolled back, it requires that the GVT computation be performed frequently in order to keep up with the real time. This can cause significant overhead. In this paper, we focus only on conservative synchronization for real-time simulation.

2.2 Interactive Simulation

By interactive simulation, we mean that a simulator needs to interact in real time with the user or with a physical system (such as a physical device and a real application). This is an area bordering both continous-time and discrete-event simulation paradigms. An important example is the Distributed Interactive Simulation (DIS) (IEEE Std 1278.1-2012, Revision of IEEE Std 1278.1-1995 2012), which has been used extensively for building computer-generated virtual environments for training and wargaming in the defense community. DIS was later superseded by the High Level Architecture (HLA) (IEEE Std 1516-2000 2010) with a sharpened focus on simulation interoperability. An important aspect of HLA is the time management of the federates, called the runtime infrastructure (RTI) (Fujimoto and Weatherly 1996).

An interesting HLA/RTI time management mechanism is based on wall-clock time, where the simulator derives the current simulation time directly from real time as opposed to using timestamps. "Real-time scheduling" in this sense refers to creating effective mechanisms, such as synchronizing hardware clocks and applying dead reckoning techniques, so that the simulator is able to meet the soft or hard real-time constraints to ensure timely interactions with the end user (human-in-the-loop) or the physical system (machine-in-the-loop). Our approach is different in that we keep stringent timestamp ordering of the events. This is especially important to incorporating analytical models, in which case we need to maintain the necessary simulation accuracy with a fine-grained timing requirement. For example, the transactions in a detailed network model may happen at or below the microsecond level. It is therefore insufficient to simply conduct a time-driven simulation based on the wall-clock time at a granularity in the milliseconds. Our real-time scheduling mechanism proposed in this paper aims at enabling interactive simulation with potentially large-scale and yet fine-grained models, such as the network simulation, where the models can be executed in parallel and faster than real time.

2.3 Network Emulation and Interactive Network Simulation

An important area of interactive simulation is network emulation. A major distinction between simulation and emulation in this case is the treatment of time. Simulation provides models for network protocols and network entities (networked hosts, routers, links), and represents network transactions as pure logic operations, where simulation time bears no direct relationship to the wall-clock time. Emulation, on the other hand, directly interacts with the physical devices by conducting and modulating traffic in real time.

Choices are abundant for network emulation. We can only mention a few prominent examples. Dummynet (Rizzo 1997) represents each virtual network link as a queue with specific bandwidth and delay constraints; network packets are intercepted in real time and then pushed through the corresponding queues in order to experience proper packet delays and packet losses. ModelNet (Vahdat et al. 2002) uses parallel computers to run a large number of unmodified network applications communicating over a virtual network. EmuLab (White et al. 2002) is an experimentation facility built on a dedicated high-performance computing cluster consisting of computers where the user can swap in specific operating systems and virtual machines to run applications; the computers are configured to form a virtual network environment with designated delay nodes for modulating traffic. ORBIT (Raychaudhuri et al. 2005) is an open large-scale wireless network emulation testbed that uses real wireless transmissions.

Most network simulators are embedded with real-time simulation capabilities to support emulation. Again, we can name only a few examples: NSE (Fall 1999), IP-TNE (Simmonds et al. 2000), Maya (Zhou et al. 2004), TWINE (Zhou et al. 2006), PRIME (Liu et al. 2007), CORE (Ahrenholz et al. 2008), PrimoGENI (Van Vorst et al. 2011), and S3F (Nicol et al. 2011). Our research here aims at addressing the general fundamental issues related to real-time processing in parallel simulation. The results can be readily used to improve the real-time performance of the parallel network simulators.

Xu et al. (2001) earlier introduced the concept of *real-time lookahead* that can be extracted from physical implementation of network protocols and applications. They observe that there is a natural delay in real time for incoming network traffic between it is generated by real applications and presented to the real-time simulator, which is due to network latency and queuing. This delay can be used as a lookahead and as a cushion for the simulator to advance its simulation clock. Note that "lookahead" here is not used in the traditional sense of quantifying the model's ability to predict future, which we normally use to guarantee the correctness and performance of parallel simulation. Here it actually reflects the quickness of the simulator's response to external events. In other words, the lack of real-time lookahead does not mean that the simulator cannot advance its clock; on the contrary, the simulator should in the least be able to advance its clock according to real time. Framing it as lookahead as a way to limit real-time event processing seems to be an unnecessarily heavy-handed approach. Different from Xu's approach, we make an important distinction between timeliness and responsiveness of a real-time simulator. The former is a measurement of quality on the design and implementation of the real-time simulator, while the latter is a user-configurable model-specific parameter. We further elaborate this point in next section.

In our earlier work, RINSE (Liljenstam et al. 2005), we made the distinction between simulated and emulated events, and used a priority-based scheduling mechanism to prioritize emulated events. However, all events are still pinned down to real time. Also, the definition of emulated events was restricted to only incoming events from the physical system; the outgoing events could not be properly identified due to the limitation of the simulator API. Consequently, there is no mechanism to guarantee RINSE's responsiveness to external events. In RINSE, we also proposed a latency-hiding technique by shortening the queuing time for emulated packets in simulation to compensate for the latency experienced by the emulated packets before they enter into the simulation system. This technique is domain specific. Our approach here is applicable to general parallel simulations. Latency hiding can be easily implemented in the network model built with our simulator with the generic real-time simulation API.

Interactive simulation has been applied more broadly for modeling networks. In Genesis (Szymanski et al. 2002), network simulation is used as an integrated service for network management, planning, monitoring, and traffic engineering. In ROSENET (Gu and Fujimoto 2007), a high-performance network simulator works with a low-fidelity network emulator, which can be geographically distributed, to enable large-scale network studies. This idea has been extended in SymbioSim (Erazo and Liu 2013), where network simulation and emulation form a symbiotic relationship, through which accurate network queuing behaviors can be reproduced in emulation using real-time simulation results. Jin et al. (2012) proposed a technique of coordinating a parallel simulator with an emulation system based on virtual machines controlled by virtual time. The two systems coordinate by alternating the execution of simulation and emulation with message exchanges occurring only at the boundaries. All these techniques require the parallel simulators be able to promptly handle external events.

3 REAL-TIME SCHEDULING

3.1 Timeliness and Responsiveness

For real-time simulation, it is important to make a distinction between two different concepts: timeliness and responsiveness. *Timeliness* is a metric for determining how well a simulator is able to keep up with the real time. We define timeliness—more specifically, the lack of it—to be the difference between the simulation time that an emulated event is scheduled to be processed and the wall-clock time at which it truly happens. Due to the simulation workload and various overheads, including the cost for retrieving an event from the event list and the cost for context switching between simulation processes, the real time at which an event gets to be executed may be extended beyond the scheduled time. We use the delay to evaluate the efficiency of the simulator in terms of processing events and controlling overheads.

Unlike timeliness, *responsiveness* is a measure of the simulator's response to external input. A real-time simulator that interacts with the physical system typically would take input from the physical system, such as reacting to the user input made through a graphical user interface, or intercepting a network packet at the network device. We define responsiveness as the real-time delay between the time an event occurs in the physical system and the time the event is being processed in simulation.

The tasks of a real-time simulator are split between processing simulation events and performing I/O functions. On the one hand, it is relatively straightforward for handling the output, i.e., performing actions initiated by the simulator and realized in the physical system, such as plotting the simulation state on a graphical display, or sending out network packets through the network interface—these output actions are usually performed synchronously. On the other hand, it is not as straightforward to handle the input, i.e., performing actions initiated by the physical system and processed in simulation. These input actions need to be performed asynchronously; the simulator usually runs as a separate process from the one that generates the input at the physical system.

One solution, as in RINSE (Liljenstam et al. 2005) and some other real-time simulators, is to have the simulator probe the arrival of external events from the physical system before processing the next event. When there are available external events, the simulator will insert them into the event list using the current simulation clock as the timestamp. The drawback is that probing must be handled with care—it could add significant overhead if the probing is set to be too frequent; and yet the simulator could become sluggish if the probing interval is too long. Alternatively, one can "pace" the simulation events in which case the simulator can simply do a timed-wait on input until the time of the next event. It would require, however, that all simulation events are processed in real time so that the simulator may not advance its simulation clock ahead of the wall-clock time at which the external event happens.

It is important to observe that the responsiveness requirement depends on the model. In other words, it should be a configurable model parameter. For human-in-the-loop cases, the responsiveness typically does not need to be smaller than, say, 100 milliseconds, for the user to notice any difference. For machine- or software-in-the-loop cases, such as in the infrastructure network simulations, the responsiveness may not

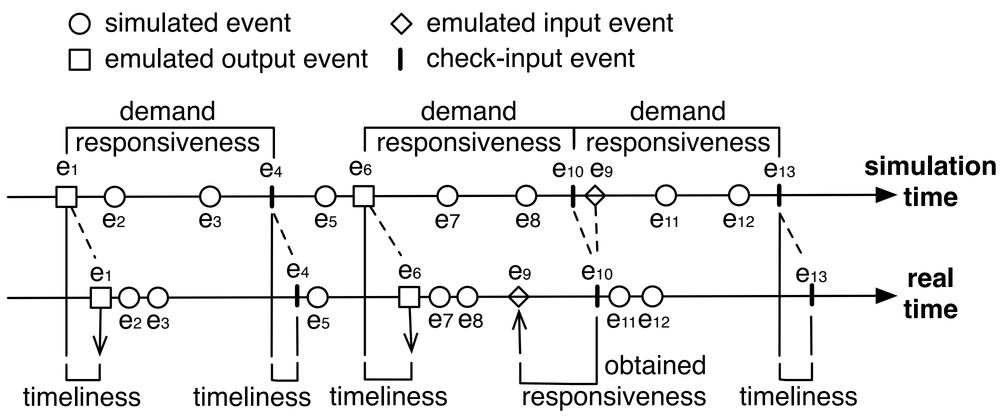

Figure 1: Simulated and emulated events.

be above the millisecond level to ensure sufficient accuracy. It can go even lower for wireless network simulations. The responsiveness value can also be determined automatically from the characteristics of the physical devices, such as the network latency and the queuing level, as what has been proposed in the real-time lookahead approach (Xu et al. 2001).

Our solution makes a deliberate distinction between simulated and emulated events. Both types of events are processed in the timestamp order at their corresponding LPs. They abide the conservative synchronization protocol; that is, to avoid out-of-order event processing, the events can only be processed if their timestamp is no larger than than the lower bound on timestamp (LBTS) determined by the synchronization protocol. A *simulated event* is processed without delay as soon as the event becomes the head of the event list. It is not pinned down to real-time. An *emulated event*, on the other hand, shall be processed according to the wall-clock time—the simulator will not process the event until the wall-clock time has reached or gone beyond the scheduled time of the event. We handle three types of emulated events:

- An *emulated output event* is an event initiated by the simulator to invoke actions in the physical system in real time. For example, one can schedule an emulated output event to report the state of the simulation or to send a network packet at a particular time.
- An *emulated input event* is an event initiated by the physical system and inserted into simulation to alter its state for modeling purposes. For example, one can insert an emulated input event upon a user input or upon a network packet being captured at the network device.
- A *check-input event* is a special event scheduled by the simulator at regular real-time intervals in the absence of other emulated events. The check-input event serves two purposes: (1) it forces the simulator to synchronize with the wall-clock time at a desirable frequency; the simulator will not be able to advance its simulation time ahead of the real time by more than the length of the time interval, and (2) it checks the input; when a check-input event is processed, the simulator will poll the physical system to see if there are emulated input events enqueued during the last interval; and if so, it will insert the events into the event list at the corresponding LP. (The simulator will automatically check input when processing all types of emulated events. The check-input event is specially designed for this purpose when no other emulated events are present.)

Fig. 1 shows an example with thirteen events at an LP's event list. The figure shows two time lines: one in simulation time, at which the events are shown to be scheduled for processing, and the other in real time, at which the same events are processed in reality. Among the thirteen events, there are six emulated events: e_1 and e_6 are emulated output events; e_9 is an emulated input event; e_4, e_{10}, and e_{13} are check-input events. The rest of the events are simulated events.

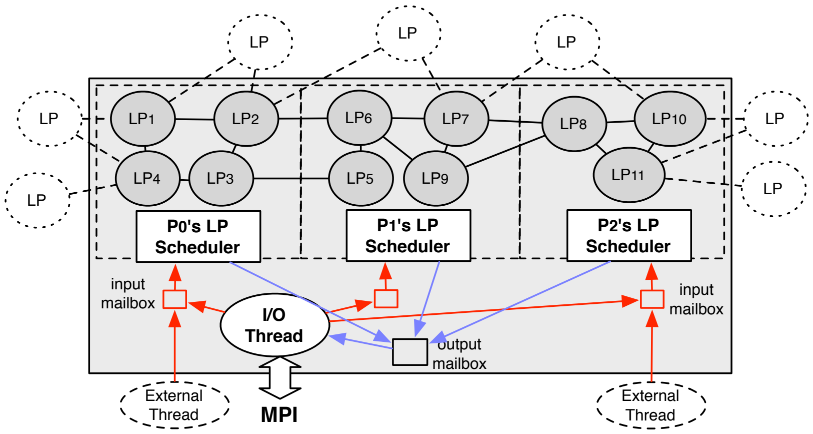

Figure 2: Real-time simulation architecture.

As seen in the example, the difference between the simulation time that an emulated event (either an emulated output event or a check-input event) is scheduled to be processed and the wall-clock time at which it truly happens is “timeliness”. Timeliness varies among the events. Collectively, it reflects how well the simulator is able to keep up with real time. The check-input events are scheduled at a regular time interval only if there are no other emulated events present during the interval. We call the length of the interval at which we schedule the check-input events “demand responsiveness”. It is the upper bound in simulation time the simulator polls the physical system for input. The example shows that an emulated input event (e_9) does not get to be processed by the simulator until the check-input event (e_{10}) is processed. The lag between the wall-clock time the event occurs in the physical system and the wall-clock time it is being processed in simulation is the (obtained) “responsiveness”. It is a measure of the simulator’s response to external input. The maximum responsiveness is determined by the demand responsiveness (i.e., the interval length) and the timeliness of the simulator. For simulated events, they are processed as soon as possible, since they do not need to be pinned down in real time.

3.2 The Real-Time Architecture

In this section, we describe the system architecture of our parallel simulator and the associated synchronization algorithm designed for supporting either real-time or proportional-to-real-time simulation execution and interaction with the physical system. Fig. 2 illustrates the components of the system. A simulation model is spatially divided into a collection of interconnected logical processes (or LPs), which can be independently assigned to different processors (or cores) on separate compute nodes for parallel processing. In our implementation, we statically assign the LPs to the compute nodes and processors. Each LP maintains an event list that stores the events associated with the LP waiting to be processed in timestamp order. The LPs communicate with one another using timestamped messages traveling through established channels between them. In the example shown in Fig. 2, eleven LPs are assigned to the local machine; they are divided to run on three separate processors.[1] Each of the three processors features an LP scheduler, which is responsible for dispatching the LPs assigned to the processor: in a loop, the LP scheduler selects an LP among the available LPs ready for execution and processes events on its event list in timestamp order according to conservative synchronization and real-time requirement. (We describe real-time LP scheduling and synchronization in the next session.)

An I/O thread is expected to run on a separate processor on the same machine. The I/O thread is responsible for communicating with the remote machines for distributed simulation. In our implementation, the I/O thread communicates with the I/O threads on other compute nodes by sending and receiving simulation

[1]We do not differentiate between processors and cores; we treat them equally as a processing unit.

events through the Message Passing Interface (MPI). To facilitate communication between the threads, we create a facility called *mailbox*. There is an input mailbox associated with the LP scheduler on each processor. Simulation events received by the I/O thread via MPI are deposited into the input mailbox of the corresponding LP. The LP scheduler later retrieves the events and then inserts them into the event list. In addition, the input mailbox is also used for accepting emulated events. An external thread run on behalf of the physical system can insert emulated events and deposit them at the corresponding input mailbox. (We discuss the real-time simulation API in section 4.)

An output mailbox is designated specifically for the I/O thread. It is used by the LP schedulers on the same machine to deposit simulation events targeting remote LPs. The I/O thread waits for the simulation events using a conditional variable; it retrieves the events from the output mailbox, serializes them into messages, and then sends them to the remote machines through MPI.

3.3 Real-Time LP Scheduling Algorithm

In the following, we discuss the real-time LP scheduling algorithm to be run at each processor. We first make a few definitions before describing the algorithm. Let ξ_{p} be the set of LPs assigned to processor p. Each LP maintains a simulation clock and an event list. Suppose e is an event on the event list, we use $\mathsf{t(e)}$ to denote the timestamp of the event. Let τ_{x} be the timestamp of the earliest event on the event list of $\mathsf{LP_x}$. If the event list is empty, we simply set τ_{x} to be infinite.

At each processor, the LP scheduler also maintains two priority queues: We use $\mathsf{R_p}$ to denote the set of LPs ready to be executed on processor p. An LP is ready to be executed if its simulation clock is less than the lower-bound on timestamp. The lower-bound on timestamp is calculated using a conservative synchronization protocol; it is the simulation time up to which the LP can process events on its local event list and advance its simulation clock without causing out-of-order event execution. We sort the LPs in $\mathsf{R_p}$ according to the timestamp of their earliest event (using a priority queue). Let $\tau(\mathsf{R_p})$ be timestamp of the earliest event of all ready LPs in $\mathsf{R_p}$. That is, $\tau(\mathsf{R_p}) = \min_{\mathsf{LP_x} \in \mathsf{R_p}}\{\tau_{\mathsf{x}}\}$.

We use $\mathsf{W_p}$ to denote the set of LPs that are ready to be run and yet are blocked due to the real-time constraint. That is, if the current wall-clock time has not reached the timestamp of the earliest event on the LP which happens to be an emulated event, the LP needs to be put on hold waiting for the real time to catch up. Similar to $\mathsf{R_p}$, we sort the LPs in $\mathsf{W_p}$ according to the timestamp of their earliest event (again, using a priority queue). We use $\tau(\mathsf{W_p})$ to denote the timestamp of the earliest (emulated) event in all LPs in $\mathsf{W_p}$. That is, $\tau(\mathsf{W_p}) = \min_{\mathsf{LP_x} \in \mathsf{W_p}}\{\tau_{\mathsf{x}}\}$.

In our implementation, we embed the real-time scheduling algorithm with a hierarchical composite synchronization protocol, which combines both window-based and CMB-based synchronization, and applies independent synchronization regimes for both shared memory and distributed memory (Liu and Rong 2012). For simplicity, here we describe the real-time scheduling algorithm using a window-based synchronization protocol (Nicol 1993). We assume the lookahead, δ, to be the minimum simulation time any LP can affect another LP. The simplified algorithm is described in Algorithm 1.

The simulation starts at time 0. We use $\mathsf{t_{sync}}$ to indicate the start of the current synchronization window, and use $\mathsf{T_{start}}$ to indicate the wall-clock time at the start of the simulation (line 1). The simulation ends when it reaches beyond the termination time $\mathsf{t_{term}}$. At the beginning of each synchronization window, all processors are engaged in an all-to-all exchange so that future simulation events (those with timestamps larger than the beginning of the synchronization window) can be delivered to the corresponding processors (line 3). The processors then perform a min-reduction to determine h, the beginning of the next synchronization window (line 4). Initially all LPs are ready to be executed (line 5). In the while-loop (at lines 6-24), the LP scheduler processes all LPs that are ready to be executed, including those currently blocked due to real-time constraints. After that, the synchronization window advances to the next (line 25).

In the while-loop, the LP scheduler first checkes the current real time $\mathsf{T_{now}}$, which is calculated by subtracting the wall-clock time at the start of the simulation, and then divided by the emulation speed-up ratio γ, which is one for real-time simulation (line 7). If the current real time has already passed the

Algorithm 1 The real-time LP scheduling algorithm on processor p

```
1: $W_p \leftarrow \phi$; $t_{sync} \leftarrow 0$; $T_{start} \leftarrow$ wallclock()
2: WHILE ($t_{sync} < t_{term}$) DO
3:     all-to-all exchange of future events among processors
4:     $t' \leftarrow \min_{LP_x \in \xi_p}\{\tau_x + \delta, t_{term}\}$; $h \leftarrow$ min-reduction($t'$)
5:     $R_p \leftarrow \xi_p$
6:     WHILE ($|R_p| + |W_p| > 0$) DO
7:         $T_{now} \leftarrow (\text{wallclock}() - T_{start})/\gamma$
8:         WHILE ($\tau(W_p) < T_{now}$) DO
9:             $LP_x \leftarrow$ delete-min($W_p$); insert($R_p, LP_x$)
10:        IF ($R_p = \phi$) THEN
11:            sleep until $T_{start} + \tau(W_p)$
12:        ELSE
13:            $LP_x \leftarrow$ delete-min($R_p$)
14:            WHILE ($LP_x$ eventlist is not empty) DO
15:                $T_{now} \leftarrow (\text{wallclock}() - T_{start})/\gamma$
16:                IF ($\tau(W_p) < T_{now}$) THEN
17:                    insert($R_p, LP_x$); break // from the event loop
18:                $e \leftarrow$ peek-min-event($LP_x$)
19:                IF ($t(e) > h$) THEN
20:                    break // end the event loop
21:                ELSE IF (e is simulated event OR $t(e) \leq T_{now}$) THEN
22:                    remove-min-event($LP_x$); process-event(e)
23:                ELSE
24:                    insert($W_p, LP_x$); break // from the event loop
25:    $t_{sync} \leftarrow h$
```

timestamp of the earliest event for those LPs blocked due to the real-time constraint, the LP scheduler unblocks them by moving them to the ready queue (lines 8 and 9). If the ready queue is empty, the LP scheduler has no more work to do other than waiting for the real time to catch up (line 11). Otherwise, an LP at the head of the ready queue (with the earliest event) is chosen to run (line 13).

In the event loop (lines 14-24), the LP scheduler takes a measure of the current wall-clock time before processing each event (line 15). We interrupt the event loop if the real time has surpassed the earliest timestamp of a blocked LP, in which case we also unblock it and move it to the ready queue (line 17). If the event at the head of the event list has a timestamp beyond the current synchronization window, we have finished processing the LP for the current round and we end the event loop (line 20). If it's a simulated event, or if it's an emulated event with a timestamp smaller than the current wall-clock time, we process the event (line 22). Otherwise, we suspend the execution of the LP due to the real-time constraint and move it to W_p for later processing (line 24).

The above real-time scheduling algorithm can be extended to be incorporated with other synchronization protocols. For example, to enable asynchronous CMB-style, one can simply add another queue to store the LPs that are blocked from execution once the simulation clock has reached the lower-bound on timestamp due to local causality constraint. These LPs will later become ready to be executed again once the lower-bound on timestamp is advanced by the synchronization protocol.

4 REAL-TIME SIMULATION API

A parallel simulator needs to provide easy and expressive program constructs for users to realize real-time (or proportional-to-real-time) simulation and seamless interaction with the physical system. Our implementation is based on the Scalable Simulation Framework (SSF), extended with the real-time support.

SSF API defines five core classes (Cowie et al. 1999). An *entity* is a container for state variables collectively representing a model component. The entities can be combined together to share the same timeline (as an LP), either explicitly by user specification or implicitly as a result of simulator's performance optimization strategy. In the model, the entities are connected by mapping the *out-channels* of the entities to the *in-channels* of other entities. The channels provide the communication end-points between the entities with specified delays. *Events* are messages sent through the channels: an event written to the out-channel will be delivered by the simulator to all mapped in-channels with proper delays. Within an entity, one can designate one or more *processes* to perform simulation activities, including receiving events at the entitys in-channels, or waiting for a certain simulation time to elapse.

For real-time simulation, we add an extension to the entity class's constructor for the user to specify an emulated entity. If an entity is emulated, all events associated with this entity will be treated as emulated events, which means they will be processed according to real time. An emulated entity can also accept input from the physical system. In the constructor, the user can specify the responsiveness of the entity for accepting external input, which can be defined as the maximum delay allowed for an entity to respond to the arrival of an external event. The simulator uses this value to define the interval for scheduling the check-input events (see section 3.1).

An external system can inject events into simulation by invoking the *insertEmulatedEvent()* method of an emulated entity. The event is then deposited to the input mailbox to be retrieved by the LP scheduler (upon processing an emulated event or a check-input event) on the processor where the corresponding entity resides. When the event is processed, a callback method, *emulate()*, of the emulated entity will be invoked to handle the event in the simulation system.

The simulation starts when the user invokes the *ssf_start()* method in the main function after the model has been created. The user specifies the simulation termination time as a parameter to the method. To enable real-time or proportion-to-real-time simulation, the user can also specify a speed-up ratio, γ, which is used to indicate the emulation speed with respect to real time, i.e., the progression of simulation time over the wall-clock time. If the speed-up ratio is one, all emulated events are expected to be processed according to real time. That is, one second of simulation time corresponds to one second in real time. If the speed-up ratio is set to be five, the simulation is expected to run five times faster than the wall-clock time. If the ratio is set to be infinite, we will run simulation independent from the wall-clock time.

5 PRELIMINARY EXPERIMENTS

We conducted preliminary experiments to examine the performance of the real-time scheduling algorithm implemented in our `minissf` simulator (MiniSSF 2013) using a simple queuing model. We run our experiments on Stampede, which is a Dell Linux cluster consisting of over 6,400 Dell PowerEdge server nodes, each with two eight-core Intel Xeon E5 processors, one Xeon Phi Coprocessor, and 32 GB memory (XSEDE 2013). For the preliminary experiments, we only conducted the experiments on 256 cores. The model consists of a number of queues connected in a circle, each being a single-server queue with infinite capacity and an exponentially distributed service time. We fix the number of queues on each core to be 100. In total, the model consists of 25,600 queues. At the start, we populate the queues with an average of 10 initial jobs, sampled from a Poisson distribution. In the addition to the two adjacent queues, each queue also connects to the third queue randomly chosen within a radius of 100. Upon the departure of a completed job at a queue, the queue sends the job to one of its connected queues with the same probability. We fix a delay of 1 ms between the departure of a job at a queue and its arrival at the next queue. We vary the mean service time of the queues in order to experiment with different simulation workload.

In the experiment, we choose one of the queues to be emulated and measure the timeliness of the simulator for processing the real-time events, including job arrival, job service, and job departure at the queue. Fig. 3 shows the measurements of timeliness of 10,000 consecutive emulated events, when the mean service time of the queues is set to be 0.4 ms. Changing the mean service time does not change timeliness

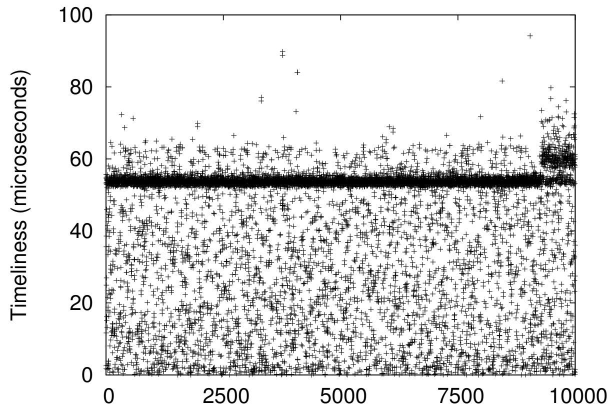

Figure 3: Timeliness.

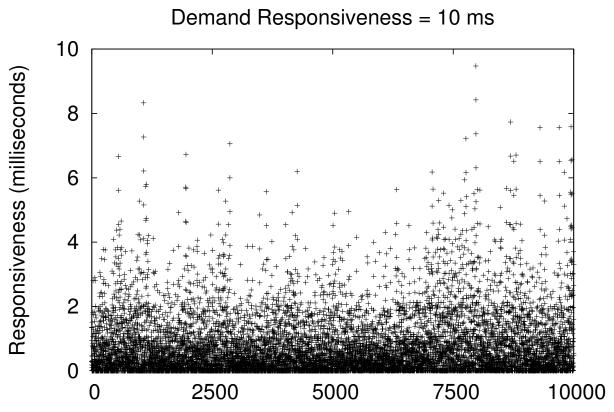

Figure 4: Responsiveness (demand=10 ms).

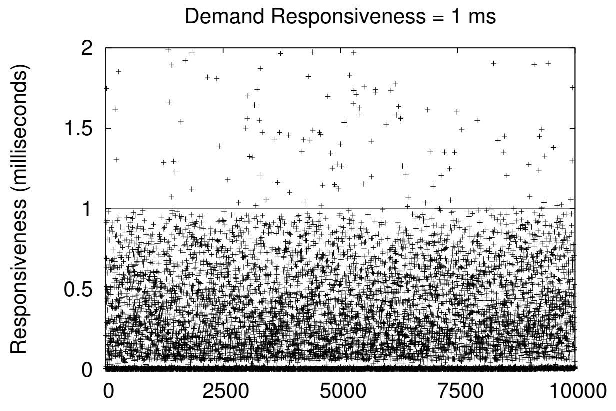

Figure 5: Responsiveness (demand=1 ms).

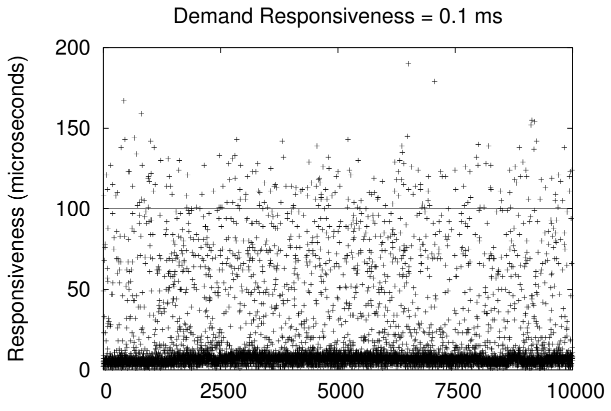

Figure 6: Responsiveness (demand=0.1 ms).

significantly as long as the simulator keeps running faster than real time. We also create a separate thread to create external events at an interval of 1 millisecond. We measure the responsiveness for processing these external events under different demand responsiveness settings (10, 1, and 0.1 ms). Figures 4-6 show the results. In most cases, the simulator is able to achieve the required responsiveness. For the 0.1 ms case, the timeliness (around 60 μs) is significant and therefore must be taken into consideration.

6 CONCLUSIONS AND FUTURE WORK

Having simulation to interoperate with physical systems in real time is important for optimizing system performance and for enabling studies with closely integrated environments for training, testing, and analytics. In this paper, we present a real-time scheduling algorithm for dealing with the mixture of simulated and emulated events in a full-fledged conservatively synchronized parallel simulation kernel. We introduce concepts important to maintaining simulation efficiency and improving timeliness and responsiveness for processing the emulated events. We propose a real-time architecture for a parallel simulator to easily and efficiently incorporate emulated events originated from the physical system. The design has been incorporated with a well-established parallel simulation API. We conducted preliminary experiments to demonstrate the real-time performance of our implementation. Future work includes more extensive performance studies of our parallel real-time simulator, and for supporting large-scale interactive network simulation and emulation studies.

ACKNOWLEDGMENTS

This research is supported in part by NSF grants CCF-0937964, HRD-0833093, IIP-1237818, and by a sub-contract from the GENI Project Office at Raytheon BBN Technologies. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), supported by NSF grant OCI-1053575.

REFERENCES

Ahrenholz, J., C. Danilov, T. Henderson, and J. Kim. 2008. “CORE: A real-time network emulator”. In *MILCOM*, 1–7.

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 (5): 440–452.

Cowie, J., D. Nicol, and A. Ogielski. 1999. “Modeling the global internet”. *Computing in Science and Engineering* 1 (1): 42–50.

Erazo, M., and J. Liu. 2013. “Leveraging Symbiotic Relationship Between Simulation and Emulation for Scalable Network Experimentation”. In *ACM SIGSIM-PADS*, 79–90.

Fall, K. 1999. “Network emulation in the Vint/NS simulator”. In *IEEE Symposium on Computers and Communications (ISCC)*, 244–250.

Fujimoto, R., D. Lunceford, E. Page, and A. M. Uhrmacher. 2002. “Grand challenges for modeling and simulation”. Technical Report 350, Schloss Dagstuhl.

Fujimoto, R., and R. Weatherly. 1996. “Time Management in the DoD High Level Architecture”. In *PADS*, 60–67.

Fujimoto, R. M. 1990. “Parallel discrete event simulation”. *Communications of the ACM* 33 (10): 30–53.

Gu, Y., and R. Fujimoto. 2007. “Applying parallel and distributed simulation to remote network emulation”. In *WSC*, 1328–1336.

IEEE Std 1278.1-2012, Revision of IEEE Std 1278.1-1995 2012. “IEEE Standard for Distributed Interactive Simulation–Application Protocols”.

IEEE Std 1516-2000 2010. “IEEE Standard for Modeling and Simulation High Level Architecture–Framework and Rules”.

Jefferson, D. R. 1985. “Virtual time”. *ACM Transactions on Programming Languages and Systems* 7 (3): 404–425.

Jin, D., Y. Zheng, H. Zhu, D. M. Nicol, and L. Winterrowd. 2012. “Virtual Time Integration of Emulation and Parallel Simulation”. In *PADS*, 201–210.

Liljenstam, M., J. Liu, D. Nicol, Y. Yuan, G. Yan, and C. Grier. 2005. “RINSE: the real-time immersive network simulation environment for network security exercises”. In *PADS*, 119–128.

Liu, J., S. Mann, N. Van Vorst, and K. Hellman. 2007. “An Open and Scalable Emulation Infrastructure for Large-Scale Real-Time Network Simulations”. In *INFOCOM*, 2476–2480.

Liu, J., and R. Rong. 2012. “Hierarchical Composite Synchronization”. In *PADS*, 3–12.

MiniSSF 2013. “Minimalistic Scalable Simulation Framework”. http://www.primessf.net/minissf/.

Nicol, D., D. Jin, and Y. Zheng. 2011. “S3F: the scalable simulation framework revisited”. In *WSC*, 3288–3299.

Nicol, D. M. 1993. “The Cost of Conservative Synchronization in Parallel Discrete Event Simulations”. *Journal of the ACM* 40 (2): 304–333.

Raychaudhuri, D., I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh. 2005. “Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols”. In *Proceedings of Wireless Communications and Networking Conference (WCNC)*.

Rizzo, L. 1997. “Dummynet: a simple approach to the evaulation of network protocols”. *ACM SIGCOMM Computer Communication Review* 27 (1): 31–41.

Simmonds, R., R. Bradford, and B. Unger. 2000. “Applying parallel discrete event simulation to network emulation”. In *PADS*, 15–22.

Szymanski, B. K., A. Saifee, A. Sastry, Y. Liu, and K. Madnani. 2002. “Genesis: a system for large-scale parallel network simulation”. In *PADS*, 89–96.

Vahdat, A., K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. 2002. “Scalability and Accuracy in a Large Scale Network Emulator”. In *OSDI*, 271–284.

Van Vorst, N., M. Erazo, and J. Liu. 2011. “PrimoGENI: Integrating Real-Time Network Simulation and Emulation in GENI”. In *PADS*, 1–9.

White, B., J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. 2002. “An Integrated Experimental Environment for Distributed Systems and Networks”. In *OSDI*, 255–270.

XSEDE 2013. “Extreme Science and Engineering Discovery Environment”. http://www.xsede.org/.

Xu, K., M. Takai, J. Martin, and R. Bagrodia. 2001. “Looking ahead of real time in hybrid component networks”. In *PADS*, 14–21.

Zhou, J., Z. Ji, and R. Bagrodia. 2006. “TWINE: A Hybrid Emulation Testbed for Wireless Networks and Applications”. In *INFOCOM*, 1–13.

Zhou, J., Z. Ji, M. Takai, and R. Bagrodia. 2004. “MAYA: integrating hybrid network modeling to the physical world”. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 14 (2): 149–169.

AUTHOR BIOGRAPHY

JASON LIU is an Associate Professor at the School of Computing and Information Sciences, Florida International University. His research focuses on parallel simulation and high-performance modeling of computer systems and communication networks. He received a B.A. degree from Beijing University of Technology in China in 1993, an M.S. degree from College of William and Mary in 2000, and a Ph.D. degree in from Dartmouth College in 2003. He served as General Chair for MASCOTS 2010, SIMUTools 2011 and PADS 2012, and also as Program Chair for PADS 2008 and SIMUTools 2010. He is an Associate Editor for SIMULATION, Simulation Transactions of the Society for Modeling and Simulation International, and a Steering Committee Member for SIGSIM-PADS. His email address is liux@cis.fiu.edu; his home webpage is http://www.cis.fiu.edu/~liux/.