The Cost of Conservative Synchronization in Parallel Discrete Event Simulations

DAVID M. NICOL

College of William and Mary, Williamsburg, Virginia

Abstract. This paper analytically studies the performance of a synchronous conservative parallel discrete-event simulation protocol. The class of models considered simulates activity in a physical domain, and possesses a limited ability to predict future behavior. Using a stochastic model, it is shown that as the volume of simulation activity in the model increases relative to a fixed architecture, the complexity of the average per-event overhead due to synchronization, event list manipulation, lookahead calculations, and processor idle time approaches the complexity of the average per-event overhead of a serial simulation, sometimes rapidly. The method is therefore within a constant factor of optimal. The result holds for the worst case "fully-connected" communication topology, where an event in any portion of the domain can cause an event in any other portion of the domain. Our analysis demonstrates that on large problems—those for which parallel processing is ideally suited—there is often enough parallel workload so that processors are not usually idle. It also demonstrated the viability of the method empirically, showing how good performance is achieved on large problems using a thirty-two node Intel iPSC/2 distributed memory multiprocessor.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: *performance attributes*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*parallel*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Conservative synchronization

1. *Introduction*

The problem of parallelizing discrete-event simulations has received a great deal of attention in the last several years. Simulations pose unique synchronization constraints due to their underlying sense of time. When the simulation state can be simultaneously changed by different processors, actions by one processor can affect actions by another. One must not simulate any element (or subsystem) of the model too far ahead of any other in simulation time, to avoid the risk of having its logical past affected. Alternately, one must be prepared to fix the logical past of any element determined to have been simulated too far.

As described in Fujimoto's excellent survey [7], two schools of thought have emerged concerning synchronization. The *conservative* school [5, 15, 25, 26]

D. M. Nicol was supported in part by the Virginia Center for Innovative Technology, by NASA grants NAG-1-060 and NAS1-18605, and National Science Foundation (NSF) grant ASC 8-19393.

Author's address: Department of Computer Science, College of William and Mary, Williamsburg, VA 23185.

employs methods that prevent any processor from simulating beyond a point at which another processor might affect it. These synchronization points need to be reestablished periodically to allow the simulation to progress. Early efforts focussed on finding protocols that were either free from deadlock, or which detected and corrected deadlock [19]. The *optimistic* school [9] allows a processor to simulate as far forward in time as it wants, without regard for the risk of having its simulation past affected. If its past is changed (due to interaction with a processor farther behind in simulation time), it must then be able to "rollback" in time at least that far, and must cancel any erroneous actions it has taken in its false future.

Conservative protocols are sometimes faulted for leaving processors idle, due to overly pessimistic synchronization assumptions. It is almost always true that individual *model elements* are blocked because of pessimistic synchronization; the conclusion that *processors* tend to be blocked requires the assumption that all model elements assigned to a processor tend to be blocked simultaneously, or that each processor has only one model element. The latter assumption pervades many performance studies, and is unrealistic for fined-grained simulation models executed on coarser grained multiprocessors. Intuition suggests that if there are many model elements assigned to each processor, then it is unlikely that *all* model elements on a processor will be blocked. Given sufficient workload, a properly designed conservative method should not leave processors idle, because there is so much work to do. Although some model elements are blocked due to synchronization concerns, other elements, with high probability, are not.

It is natural to ask how much performance degradation due to blocking a conservative method suffers. We answer that question, by analyzing a simple conservative synchronization method. The method assumes the ability to presample activity duration times [21] (or construct lower bounds on them), and assumes that any queuing discipline used is nonpreemptive. The protocol itself is quite simple. As applied to a queuing network, it works as follows. First, whenever a job enters service, the queue to which the job will be routed is immediately notified of that arrival (sometime in the future), and the receiving queue computes a service time for the new arrival. These two actions constitute *lookahead*, a concept that is key to the protocol's success. Now imagine that all events with time-stamps less than t have already been processed and that the processors are globally synchronized. For each queue we determine the time-stamp of the next job it would route (excluding one in service) if no further arrivals occur at that queue. The processors cooperatively compute the minimum such time, say $\delta(t)$. We show that all further messages to be sent in the simulation have time-stamps at least as large as $\delta(t)$. Consequently a processor may evaluate, in parallel with all other processors, all of its events with time-stamps less than $\delta(t)$. Having done so, the processors synchronize globally, and repeat the process. The interval $[t, \delta(t))$ is called a *window*, and $\delta(t) - t$ is its *width*.

We analyze the performance of the protocol by first deriving an approximated lower bound on the equilibrium mean window width. We then multiply this width by the equilibrium rate at which the simulation generates events. The resulting product is an approximated lower bound on the average number of events that are processed within a window. We then identify conditions under which the average number of events processed in a window increases

without bound as the system simulation event generation rate increases. Next we analyze the synchronization, idle time, lookahead calculation, and event-list overheads of the protocol as a function of the average instantaneous number of on-going activities T (with associated beginning and ending events) in the system. We show that idle time exacts an $O(1)$ cost, and that a lookahead calculation has the same asymptotic per-event cost as does a manipulation of an event list in a optimized serial simulation. The per-window synchronization cost does not change if the architecture remains fixed. Thus, the protocol's asymptotic performance (as $T \to \infty$) is within a constant factor of optimal. In fact, the better the lookahead (in terms of a minimum event time), the faster the synchronization and idle time costs are amortized as T grows. Finally, we demonstrate the viability of the protocol empirically. A parallel simulation system based on the protocol has been implemented on a thirty-two node Intel iPSC/2 distributed memory multiprocessor [2]. Processor efficiencies in the range of 60%–90% are reported for several different large simulation models.

It is important to remember that our ability to process events concurrently within a window rests on assumptions abut the simulation that are not always met. The concurrency we exploit follows entirely from the assumed lookahead; the protocol will not work, at least without modification, on simulation models that permit arbitrary preemption. It is also important to remember that our analysis concerns average case performance based on a general stochastic model. Specific problem examples can be constructed to ensure that the protocol essentially executes serially, while another can execute many things in parallel. We believe that such examples are somewhat artificial and do not shed a great deal of light on how performance will behave over a wide range of problems. Our intention is to study the average case performance on a model of typical simulation problems.

This paper makes two basic contributions: One is to develop a new approach for the analysis of parallel discrete-event simulations. The second is a demonstration that many large simulation models having much concurrent activity can be effectively simulated in parallel using a simple conservative protocol.

This paper is organized as follows: Section 2 gives some background for this work. Section 3 describes the model of discrete-event simulations we use in our protocol and analysis, and then introduces the protocol. Section 4 derives an approximated lower bound on the average number of events processed in a window. Section 5 determines the complexity of the average total overhead per event suffered by suing the protocol. Section 6 reports on the performance of the protocol on several different simulation models. Section 7 gives our conclusions.

2. *Background*

Our protocol is similar to others recently proposed [1, 4, 15, 16, 30]. Unlike earlier asynchronous protocols, these synchronously move a window across simulation time, roughly as follows. Let *floor* be the lower edge of the window. This means that all events with time-stamps less than *floor* have already been processed. The processors then cooperatively determine the upper window edge, *ceiling*. This value is chosen in such a way that all events within [*floor*, *ceiling*) can safely be processed in parallel. *ceiling* becomes *floor* for the next window, and so on.

The major question with synchronous conservative protocols is whether windows small enough to prevent dependencies between window events admit enough of such events to keep all the processors busy. Lubachevsky was the first to answer this question [16], by deriving a lower bound on the number of events processed within a window defined by his method. Using this bound and some assumptions concerning event density (in simulation time), he showed that the performance of this method scales up as the problem size and number of processors are simultaneously increased. However, his results are not quantitative, although they might have been so developed. Our analysis is different in that we define a model from which event densities follow naturally, and we quantify the average number of events processed in a window. Our is an average-case analysis, while Lubachevsky's is a worst-case analysis. Also, Lubachevsky's analysis hinges on the assumption of a nonzero minimal propagation delay, while ours does not. We *do* show that minimum duration times can dramatically improve the average number of events processed within each window.

The protocol we study is an application of the one described by Chandy and Sherman [4] to a more restricted problem domain. Like Lubachevsy's method, Chandy and Sherman's application requires periodic global synchronization among processors. In each window, their protocol computes the minimum time-stamp among all "conditional" events, and then processes all "unconditional" events with smaller time-stamps. In addition, their technique incorporates the conversion of "conditional" events into "unconditional" events, as a function of messages exchanged in the simulation. Such conversion is highly application-dependent. The most important difference between our protocol and the general conditional-event approach lies in the specificity of our conversion of conditional events into unconditional events, in a way that requires little model-specific information. Furthermore, our protocol is stated within the context of a model closer to those used by simulation practitioners than is the model used to describe the conditional-event approach.

Our analysis of lookahead is related to that developed by Lin and Lazowska in [12], and by Wagner and Lazowska in [32]. Their work analyzes the ability of different queue types to predict future behavior and focuses on lookahead at a single queue. Our analysis is of a much simpler lookahead scheme, but one that is analyzed over the entire simulation. The protocol we describe can be easily adapted to accommodate these more complex techniques for computing lookahead. We have also analyzed a different class of simulations than the one studied here, on massively parallel architectures [22]. The sensitivity of performance to lookahead is quantified, upper bounds on optimal and optimistic performance are derived, as is a lower bound on the performance of the same protocol we study in this paper.

Some analysis exists of the optimistic "Time Warp" method of synchronization. The earliest analyses concerned detailed stochastic models of two processor systems [11, 20]. These models include overhead costs and permit heterogeneous processors. Most other studies of Time Warp tend to assume negligible state-saving and rollback costs. For example, Lin and Lazowska have shown that if Time Warp has no state-saving or rollback costs, and if "correct" computations are never rolled back, then Time Warp achieves optimality [13]. This is intuitive, because Time Warp aggressively searches for the simulation's critical path—if it is able to do so without cost, its performance must be

optimal. Other analyses highlight the fact that Time Warp can "guess right" while conservative methods must block. Lipton and Mizell have shown that there is a certain asymmetry between optimistic and conservative methods: while it is possible for an optimistic method to arbitrarily outperform a conservative method, the converse is not true [14]. Their analysis explicitly includes overhead costs. Madisetti, et al. [18] have developed a performance model that estimates the rate at which simulation time advances under an optimistic strategy such as Time Warp. They model the behavior of the system as a Markov chain, and include the cost of communication and of synchronization. Their analysis is exact for two processors, and approximate for a general number of processors. Lubachevsky, et al. use a sophisticated stochastic model to show how it is possible for Time Warp simulations to thrash in periods of "cascading rollbacks" [17]. Finally, Gupta, et al. have constructed an approximate model of Time Warp that has been validated on an actual Time Warp implementation [8].

3. *Model and Protocol*

We now describe our model of discrete-event simulations more formally, and define the synchronization protocol. In the discussions to follow every reference to "time" will be implicitly to simulation time; any reference to physical time will be explicitly described as such.

3.1. MODEL ASSUMPTIONS. Consider a domain composed of *sites*; each site contains a number of *servers* where *activities* occur. A site should be viewed as an atomic simulatable unit, a logical collection of entities (servers) that are simulated together. For example, a G/G/k queue is modeled as a site with k servers, and a job's receipt of service is an activity. One groups the servers together into a site because they share a common queue. The binding of activities to servers is one aspect of a site's logical role. Our discussion assumes that each site maintains its own event list, which contains all events related to servers stationed at the site.

An activity begins, ends, and (upon its completion) enables (i.e., causes) other activities. These causations are reported to the appropriate sites (and hence their servers) by way of *completion* messages. Consequently, three distinct events are associated with each activity: enable, begin, and complete. The enable event for a given activity can be different from the begin event if the site has a finite number of servers, and so imposes queuing. We permit a completion to cause more than one activity, in order to include simulation problems such as Petri-nets where a single transition firing may cause token arrivals at multiple Petri-net places. Thus, we assume that a complete event at a site causes an activity at each member of a random subset of other sites. All enable events caused by a completion have the same time-stamp as the completion.

An activity is said to be *occurring* at time t if its associated begin event has a time-stamp no greater than t, and its complete event has time-stamp no less than t. Each site maintains its own priority queue of events associated with activities enqueued or occurring at the site. Each site also maintains its own simulation *clock*, which records the time-stamp on the last event processed.

Under the assumptions of our model the enable and complete events are *unconditional*—once placed on the event list, no further activity in the simula-

tion will change them. begin events may be conditional. For example, a begin event at time t might describe the future placement of a particular job into service at a queue at time t. If before t another job with higher priority arrives, that begin event may be removed from the event list.

Depending on the ability of the site, activities may occur there one at a time, or concurrently. In the latter case, the site may have a finite number of servers; alternately, if activities never queue at a site we will say it has an *infinite capacity server*. We assume that a server is not left idle if there is an enabled activity at its site to which it can give service.

The delay in simulation time between when an activity begins and ends is called its *duration*. We assume that a duration is strictly positive, but do not assume a minimal duration. For the purposes of analysis, we assume that the simulation model is ergodic, and that each duration time comes from a distribution composed by adding a nonnegative constant to an exponentially distributed random variable. The constant models the fact that many activities one simulates have a fixed startup cost, for example, chocking a bit in a drill press prior to simulating a variable time drilling task. As is often the case in performance analysis, we assume the exponential distribution because its residual life distribution is memoryless. The correctness of the protocol does not depend on either of these assumptions. Each site may have a unique distribution, but all servers at a site must share the same distribution. A server with infinite capacity gives identically distributed service to each activity occurring at its site.

Our performance analysis rests on a number of assumptions about the simulation model which are exploited by the protocol. We assume that:

(1) Once an activity begins, the causation of further activities cannot affect its completion time.
(2) The simulation state change due to an activity completion is very local—the state change is implied by knowledge of which activity completed, which activities are subsequently caused, and the time of the completion.
(3) The activities caused by the completion of activity A_j can be reported to their respective sites during (in wallclock time) the processing of A_j's begin event. We call this the *pre-sending* of the completion message.
(4) A lower bound on the duration of an activity can be determined at the wallclock time of the receipt of the completion message that causes the activity.

To illustrate these assumptions, consider a job J which at time s begins service at a nonpreemptive queue Q_1, completes at time $\hat{s}$, and is routed to Q_2. Assumption 1 is satisfied by the nature of Q_1's queuing discipline. Assumption 2 is satisfied because the change in model state due to this departure is completely characterized by knowledge of Q_1, Q_2, and $\hat{s}$. Assumption 3 is satisfied if the service discipline at Q_1 is non-preemptive and the routing is stochastic (and is therefore independent of the jobs enqueued at time $\hat{s}$): in the simulation we can report the arrival of J at time $\hat{s}$ to Q_2 while processing the entry of J into service at Q_1, at time s. By doing so, the processing required of J's completion event at $\hat{s}$ does not include reporting J's departure, but may include the recording of statistics which depend on all simulation activity at Q_1 (including arrivals) up to time $\hat{s}$. Assumption 4 is satisfied if J's service time at Q_2 can be computed at the time that Q_1 reports the arrival of J to Q_2. This is

possible if the service time of every job at Q_2 is drawn independently from the same probability distribution.

This model describes a large number of common simulation models, and is related to *event graphs* described in [28] and [29]. Many queuing networks are obviously captured. Logic networks are described, with activities corresponding to logical module evaluations. Here new activities are caused when a module output changes state. *Lookahead* plays a major role in our synchronization method and its analysis. Lookahead exists and is exploited by assumptions 3 and 4 above. As we will see in the discussion of a logic network simulation in Section 6, less obvious interpretations of "activities" permit the parallelization of additional models where there is lookahead in the form of minimal nonpreemptive service times, but others of these model assumptions are not met (e.g., (1) and (3) above). In addition, there are results reported in Section 4 for models where we assume only that all duration times are bounded from below by some $D_{\min} > 0$ and that all completion messages are present.

Simulation workload is the event processing. This includes changing anticipated event times as a result of newly caused activities, in changing simulation state variables, and in gathering/recording statistics. We view event list management costs as inescapable overheads associated with the processing events.

3.2. PROTOCOL DEFINITION. Next we define the synchronization protocol in terms of the model given in 3.1. Our only architectural assumptions are that the simulation model is executed on a multiprocessor having P processors; any processor can send a message (indirectly, if needed) to any other processor, and the processors can synchronize globally.

One important aspect of our protocol is the presending of completion messages (see assumption 3 above). Let A_j be some activity whose begin event has time-stamp s. Let $\hat{s}$ be A_j's completion time. Under our protocol A_j's server must send completion messages to all sites where activities caused by A_j's completion will occur, *during the processing of* A_j*'s* begin *event*. Observe that even though the simulation time at A_j's server is s, these completion messages are time-stamped with time $\hat{s} > s$. A site which receives such a notification inserts an enable event with time-stamp $\hat{s}$ into its event list (a site with an infinite capacity server may directly insert a begin event with time $\hat{s}$); it also selects a duration time (or a lower bound on it) for the newly caused activity.

Suppose the processors have globally synchronized, and let t be the minimum time-stamp among events at all sites. Each server S_i can determine a lower bound $\delta_i(t)$ on the earliest completion time of any of its pending (i.e., as yet not begun) activities, assuming no further begin events are assigned to it. We call this the server's *lookahead bound*. For example, consider a server S_i at a single-server site. There are three cases to consider.

Case 1. *The event list at* S_i*'s site is void of* enable *events*. In this case, we define $\delta_i(t) = \infty$.

Case 2. *No activity is occurring at time* t, *and event list at* S_i*'s site contains* enable *events*. Assuming no that further enable events will be inserted into the event list, define $\delta_i(t)$ to be the completion time of the next activity to be given service.

Case 3. *Some activity is occurring at t, and event list at S_i's site contains* enable *events.* Assuming that no further enable events will be inserted into the event list, define $\delta_i(t)$ to be the completion time of the next enabled activity to receive service after the current one completes.

We can view a site with an infinite capacity server as having as many servers as it has occurring activities. In this case, we abuse notation slightly and take $\delta_i(t)$ to be the minimum lookahead bound among all these "servers". Two cases may arise. If there are no begin events in the site's event list, then define $\delta_i(t) = \infty$. If there are begin events in the site's event list, define $\delta_i(t)$ to be the minimum completion time among these.

Now consider a multi-server site. One constructs the value $\delta_i(t)$ for each server at the site by assuming no further activities will be caused at that site, permitting one to then assign the known as-yet-nonoccurring activities to servers in accordance with the site's assignment rules. This essentially defines a queue for each server; the $\delta_i(t)$ construction rules for single servers are then used for each server.

Finally, define

$$\delta(t) = \min_{\text{all servers } S_i} \{\delta_i(t)\}.$$

The protocol is very simple. Define $w_1 = 0$, and proceed as follows:

(1) Given w_n, the processors cooperatively determine $\delta(w_n)$.
(2) Each site may be simulated in parallel with all others until the time of the event with least time-stamp at that site is as large as $\delta(w_n)$. The processing of any begin event in this interval must include pre-sending the associated completion messages.
(3) Sites receive the messages sent during the processing of $[w_n, \delta(w_n))$, select duration times for the associated caused activities, and insert events into their event lists.
(4) $n = n + 1$. Goto step (1).

The obvious question to ask of this protocol is whether the sites can safely process all events within a window. The protocol is safe if, once the window is established, no further messages with time-stamps less than the upper edge of the window will ever be sent. The following theorem establishes this fact:

THEOREM 3.1. *Let $[w_n, \delta(w_n))$ be a window established by the protocol. Then every completion message sent during the processing of $[w_n, \delta(w_n))$ has a time-stamp at least as large as $\delta(w_n)$.*

PROOF. Completion messages are pre-sent by the processing of begin events. Let $b_0, \ldots, b_k$ be the times of all begin events in $[w_n, \delta(w_n))$, in increasing order. We use induction to show that for $i = 0, \ldots, k$, the completion messages associated with the begin event at time b_i have time-stamps at least as large as $\delta(w_n)$. For the base case consider b_0, and let S_i be the associated server. S_i computes $\delta_i(w_n)$ to be the minimum time-stamp on the next message it sends, *provided no further messages are received at S_i's* site. By construction, S_i's site will not receive any further messages with time-stamps less than b_0; therefore, the decision to begin the activity at b_0 was correctly foreseen during the computation of $\delta_i(w_n)$, implying that the completion time

of the activity beginning at b_0 is no smaller than $\delta_i(w_n)$, and hence is no smaller than $\delta(w_n)$. This establishes the base of the induction. For the induction step, suppose that the completion times of the activities begun at times $b_0, \ldots, b_{j-1}$ are all no smaller than $\delta(w_n)$. Consider the activity begun at time b_j, and let S_i be its server. As a consequence of the induction hypothesis, during the processing of $[w_n, \delta(w_n))$ S_i's site cannot receive any messages with time-stamps less than b_j. Consequently, the decision to begin an activity at time b_j was correctly foreseen during the computation of $\delta_i(w_n)$. The completion time of the activity beginning at b_j is thus no smaller than $\delta_j(w_n)$, and so is no smaller than $\delta(w_n)$. This completes the induction. □

Under the assumption of non-zero duration times, it will always be true that $w_n < \delta(w_n)$. Consequently, simulation time advances each window (even if no events occur in the window), and deadlock never occurs.

3.3. EXAMPLE. An example helps to further illustrate the protocol's mechanism. Consider a system with two sites $Site_1$ and $Site_2$. $Site_1$ permits an unbounded number of activities to occur simultaneously, while $Site_2$ has only one server. The system moves *objects* between sites. Duration times are random. When an object completes its duration it either disappears, moves to another (possibly the same) site, or splits into a number of objects that move. $Site_2$ uses Last-Come-First-Serve (LCFS) queuing.

Let $w_n = 100$, and imaging that objects O_1 and O_2 are present at $Site_1$, with scheduled completion times of 100 and 103. Object O_3 is in service at $Site_2$, and will complete at time 101. Object O_4 is enqueued at $Site_2$, and will eventually receive four units of service.

The completion of O_1 at time 100 sends O_1 back to $Site_1$, where it will receive another 8 units of service; the completion of O_2 at time 103 sends O_2 to $Site_2$ where it will eventually receive 6 units of service; O_2's completion at time 103 also creates a new object O_5 which is sent to $Site_1$, where it receives 4 units of service. At site $Site_2$, O_3 completes at time 101, and then remains at $Site_2$, where it will receive another 5 units of service. Observe that the messages reporting the completions of O_1, O_2, and O_3 have already been sent, and the "next" durations of those objects have already been chosen.

This scenario is summarized in figure 1, along with the contents of $Site_1$ and $Site_2$'s event lists as observed at time 100. These event lists reflect the practice of pre-sending object arrival notices. $Site_1$ determines its lookahead bound $\delta_1(100)$ by finding the minimum completion time among all objects it knows will arrive at or after time 100. O_2 arrives (again) at time 100 and completes at 108. O_5 arrives at 103 and completes at 107, making $\delta_1(100) = 107$. $Site_2$ determines $\delta_2(100)$ by identifying the next object to complete service that isn't already in service. Because $Site_2$ is LCFS, the arrival of O_3 at time 101 causes O_3 to receive service before O_4. $\delta_2(100)$ is 106, so that $\delta(100) = 106$. $Site_1$ and $Site_2$ are thus free to simulate all events with times no greater than 106, in parallel. $Site_1$ has four such events, $Site_2$ has three (or four, if the processing of the O_3 arrival event at 101 creates a begin event at 101).

Arrival events (enable events) at $Site_1$ may also serve as begin events since no queuing is imposed. Each site's processing of arrival events includes the decision of where to route the object upon completion, and the generation of completion messages with the appropriate time-stamp.

Objects at time 100

Object	Site	Arrives	Duration	Completes	Routed	Comments
O_1	$Site_1$	?	?	100	$Site_1$	Occurring at time 100
O_1	$Site_1$	100	8	108	?	Caused by completion of self
O_2	$Site_1$	?	?	103	$Site_2$	Occurring at time 100
O_2	$Site_2$	103	6	?	?	Caused by completion of self
O_5	$Site_1$	103	4	107	?	Caused by O_2 at $Site_1$
O_3	$Site_2$	?	?	101	$Site_2$	Occurring at time 100
O_3	$Site_2$	101	5	106	?	Top priority activity at 101
O_4	$Site_2$	?	4	?	?	Lower priority at 101

$\delta(100) = 106$

Event Lists

$Site_1$ Event List at time 100		$Site_2$ Event List at time 100	
Event	*Time*	*Event*	*Time*
O_1 completes	100	O_3 completes	101
O_1 arrives	100	O_3 arrives	101
O_2 completes	103	O_2 arrives	103
O_5 arrives	103		
4 events processed in [100, 106)		3 events processed in [100, 106)	

FIG. 1. Example of synchronous protocol operation.

3.4. CORRECTNESS OF LOOKAHEAD BOUNDS. It may happen that the lookahead bound $\delta_t(w_n)$ for a server S_t is "incorrect" in the sense that the next message S_t eventually sends may have a time-stamp less than $\delta_t(w_n)$. This is possible because the lookahead bound is predicated on the assumption that no as-yet-unreported activities will be caused at S_t's site. If further activities *are* caused during the processing of $[w_n, \delta(w_n))$, then the next message eventually sent by S_t may have a time-stamp less than $\delta_t(w_n)$. The important point is that the window is constructed so that every processor accurately knows its message history during $[w_n, \delta(w_n))$(this was established by Theorem 3.1), and therefore will not incorrectly execute any event. This property holds regardless of whether $\delta_t(w_n)$ accurately bounds the time of S_t's next message.

Consider the following example. At $t = 90$ a LCFS queue Q_1 may have a job in service which will complete at time 100, and one known arrival at time 98 with a service requirement of 10. The queue's lookahead bound is $\delta_1(90) = 110$; let us suppose that $\delta(90) = 95$. Now imagine there is another queue Q_2 that puts a job J into service precisely at time 91, that J's service duration at Q_2 is 5, and that J is destined to be routed to Q_1. J's begin event is processed during window [90, 95), and notification of its arrival at Q_1 is sent and received before the following subsequent window is defined. As a function of computing $\delta_1(95)$, Q_1's site will note J's arrival at time 96, and infer that J will be the next job to receive service at Q_1 (in the absence of further arrivals). Q_1 may decide to give J only 1 unit of service, giving it a departure time of 101. Thus the next message sent by Q_1 after time 90 has a time-stamp significantly less than $\delta_1(90)$. However, the protocol permits Q_1 to realize this "error" during the construction of $\delta_1(95)$.

Thus, even though the protocol is correct one should not assume that every lookahead bound computed by a every server is correct. However, the proof of Theorem 3.1 does demonstrate that if server S_i begins an activity during the processing of window $[w_n, \delta(w_n))$, then $\delta_i(w_n)$ is correct.

4. *Analysis of Protocol*

Our performance analysis derives an approximated lower bound M' on the true mean window width M, then multiplies by the equilibrium event creation rate in order to bound the average number of events created per window. By flow balance, this equals the average number of events processed within M' units of simulation time, and hence is a lower bound on the average number of events processed in M units of simulation time. We then consider the behavior of this average as a function of simulation activity rate, and minimum duration time.

The analysis to follow uses results from the theory of stochastic order relations, and manipulates hazard rate functions. Readers unfamiliar with these tools should consult Ross [27]; the appendix quickly sketches the main ideas and results we use.

We are interested in the limiting value of the expected window width $E[\delta(w_n) - w_n]$ as $n \to \infty$, supposing that the limit exists. As we will see, a window's width is comprised of the minimum of a number of complicated random variables. Complications arise both due to randomness in the model (e.g., random selection of sites where activities are caused following a completion), and due to dependence of the random variables' distributions on the past activity in the simulation. Our approach is to bound the mean window width from below with the mean minimum of much simpler, and stochastically smaller, random variables. The stochastically smaller variables are constructed by considering hazard rate functions. This is a useful analytic trick that exploits the fact that the hazard rate function for the minimum of a group of independent random variables is just the sum of their individual hazard rate functions.

One step in the bounding argument is intuitive, but not rigorously justified. Therefore, one can only rigorously call our results approximate.

The analysis uses a slightly more formal model than we have yet described. The duration time distribution for server S_i is taken to be $D_i + \exp\{\mu_i\}$, where $D_i \geq 0$ is constant and $\exp\{\mu_i\}$ is exponential with mean $\mu_i = 1/\lambda_i$. We let $D_{\min}$ be the minimum D_i value among all servers. The discussion of random variables, means, and hazard rates all concern the stochastic portion of the duration times.

Our bounds depend on the manner in which a completing activity causes activities elsewhere. To more precisely describe these effects, for every server S_i let $Reach(S_i)$ be the set of all sites where activities caused by a completion at S_i can occur. For convenience, we assume that the activities caused by a single completion are all at different sites. Activity A_j completing at S_i randomly chooses a subset $B_j \subseteq Reach(S_i)$, and causes one activity at each site in B_j. We assume B_j is chosen independently of the duration values of the caused activities. The distribution governing this choice is particular to S_i; $p(B,i)$ denotes the probability that $B \subseteq Reach(S_i)$ is the selected set.

Let A_j be an activity occurring at server S_i, and let B be the set of sites with activities caused by A_j. We are interested in the rate at which the first activity

completes, among all those caused by A_j. Towards this end, we focus on the stochastic portion of these activity durations. The "rate" of the minimum stochastic portion is just $\lambda_B = \Sigma_{S_m \in B} \lambda_m$ (see Section A.2). The expected rate (with respect to the distribution of B) is defined by

$$\psi_i = \sum_{B \subseteq Reach(S_i)} p(B, i) \left(\sum_{S_m \in B} \lambda_m \right)$$

$$= \sum_{S_m \in Reach(S_i)} \Pr\{\text{completion at } S_i \text{ causes an activity at } S_m\} \lambda_m. \quad (1)$$

Pathological analytic difficulties are avoided by assuming that the simulation model always has at least one activity occurring that causes other activities. This can be ensured, for example, by adding a "clock" site that does nothing but process a single, periodically self-causing activity.

Let $\mathscr{A}(w_n)$ be the random set of activities occurring at time w_n. During most of the analysis to follow, we condition on knowing that $\mathscr{A}(w_n)$ is some fixed subset V. The conditioning is undone later when we take an expectation with respect to the distribution of $\mathscr{A}(w_n)$.

The discussion to follow focuses on activities. To facilitate precise reference, we often use the notion $S_{s(j)}$ to describe the server at which activity A_j occurs. We want to distinguish sites with infinite capacity servers from those with a finite number of servers. We therefore define the indicator coefficient γ_i to have value 0 if the server S_i has infinite capacity, and to have value 1 if S_i's site has a finite number of servers.

For every server S_i, $\delta_i(w_n)$ is computed to be the completion time of some activity to which S_i anticipates giving service, say $Q_i(w_n)$, but which is not yet occurring a time w_n. That is,

$$\delta(w_n) = \min_{\text{all servers } S_i} \{\text{Completion time of } Q_i(w_n)\}.$$

Since activity causations become known only by the act of processing a begin event, it follows for each activity $Q_i(w_n)$ that the activity which causes $Q_i(w_n)$ is occurring at time w_n, or has already completed by time w_n. In the latter case, at the time w_n $Q_i(w_n)$ must be enqueued since S_i is busy; barring further arrivals at S_i, $Q_i(w_n)$ enters service immediately upon the completion of this activity. Thus, for every $Q_i(w_n)$, there corresponds a unique activity, say $P_i(w_n)$, that is occurring at time w_n and which *precedes* $Q_i(w_n)$ in the sense that its completion permits the evaluation of $Q_i(w_n)$'s begin event. We define $Precede(w_n)$ to be the set of all such "preceding" activities. Observe that if $Q_i(w_n)$ was enabled before w_n then at time w_n $P_i(w_n)$ is occurring at S_i; $P_i(w_n)$ may otherwise be occurring at any server that can cause events at S_i's site.

Each $\delta_i(w_n)$ may thus be written as the sum of the completion time of $P_i(w_n)$ and the duration time of $Q_i(w_n)$:

$$\delta(w_n) = \min_{\text{all servers } S_i} \{\text{Completion time of } P_i(w_n) + \text{Duration of } Q_i(w_n)\}.$$

We can also compute this minimum from the point of view of the activities in $Precede(w_n)$. An activity $A_j \in Precede(w_n)$ may precede an activity at A_j's server, any number of activities at other servers, or both. Define $Follow(A_j)$ to be the set of activities A_j precedes, and let $R_j(w_n)$ denote the *residual*

duration time of A_j—the difference between A_j's completion time and w_n. The completion time of any activity in $Follow(A_j)$ can be written as the sum of its duration, and $w_n + R_j(w_n)$. We may thus write

$$\delta(w_n) = \min_{A_j \in Precede(w_n)} \left\{ \min_{Q_k(w_n) \in Follow(A_j)} \{\text{Completion Time of } Q_k(w_n)\} \right\}$$

$$= \min_{A_j \in Precede(w_n)} \left\{ w_n + R_j(w_n) + \left\{ \min_{Q_k(w_n) \in Follow(A_j)} \{\text{Duration of } Q_k(w_n)\} \right\} \right\}. \tag{2}$$

We can construct a lower bound on the minimum completion time among all members of $Follow(A_j)$ by observing that $Follow(A_j)$ is a subset of the union of $Q_{s(j)}(w_n)$, and the set of all activities caused by A_j's completion. Taking the minimum completion time over this larger set establishes the bound. Now let $N_j(w_n)$ be the duration time of $Q_{s(j)}(w_n)$, let B_j be the set of sites where A_j's completion causes activities, and let $\{D_k + Y_{j,k} | Site_k \in B_j\}$ be the set of durations of these activities. For every $A_j \in Precede(w_n)$, we can construct the *lookahead bound* $K_j(w_n)$ to bound from below the earliest completion time of any activity in $Follow(A_j)$:

$$K_j(w_n) = w_n + R_j(w_n) + \min\left\{ N_j(w_n), \min_{Site_k \in B_j} \{D_k + Y_{j,k}\} \right\}.$$

Note that this definition can be applied to any activity occurring at time w_n. Finally, observe that the minimum taken over all $A_j \in Precede(w_n)$ in eq. (2) cannot increase if taken instead over the larger set V of all activities occurring at time w_n. This observation yields the bound

$$\delta(w_n) \geq \min_{A_j \in V} \{K_j(w_n)\}$$

$$= w_n + \min_{A_j \in V} \left\{ R_j(w_n) + \min\left\{ N_j(w_n), \min_{Site_k \in B_j} \{D_k + Y_{j,k}\} \right\} \right\},$$

from which follows a bound on the expected window width

$$E[\delta(w_n) - w_n | \mathscr{V}(w_n) = V]$$

$$\geq E\left[\min_{A_j \in V} \left\{ R_j(w_n) + \min\left\{ N_j(w_n) \min_{Site_k \in B_j} \{D_k + Y_{j,k}\} \right\} \right\} \right]. \tag{3}$$

As an important aside, note that when $D_k \geq D_{\min} > 0$ for all sites $Site_k$ we may use inequality (3) to establish the distribution-free bound

$$\delta(w_n) - w_n \geq D_{\min}. \tag{4}$$

This follows by setting all stochastic components of the right-hand-side of inequality (3) to zero, and factoring out $D_{\min} \leq D_k$ and $D_{\min} \leq N_j(w_n)$. From the definitions of $N_j(w_n)$ and $D_k + Y_{j,k}$ we see that for this inequality to hold we require only that every activity duration be at least $D_{\min}$, and that completion messages be pre-sent. We later use this result to establish distribution-free performance bounds on the protocol. The analysis to follow permits sharper bounds to be constructed, especially if $D_{\min} = 0$.

The expectation in the right-hand-side of inequality (3) is complicated by its dependence on the history of the synchronization behavior up to time w_n. For example, suppose that activity A_j began in the $(n - b_j)$th window, for some $b_j > 0$. The distribution of $K_j(w_n)$ must be conditioned on the event $\mathscr{G}_j(w_n)$ that $K_j(w_{n-c}) \geq w_{n-c+1}$ for all $1 \leq c < b_j$. Since $K_j(w_n)$ is largely comprised of random variables that also comprise $K_j(w_{n-c})$ for each c, conditioning on $\mathscr{G}_j(w_n)$ makes each $K_j(w_n)$ probabilistically larger than it would be if each component random variable had its original, unconditional distribution. The starting point for our bound is to build a stochastically smaller replacement for each $K_j(w_n)$ by replacing each of $K_j(w_n)$'s components with a pristine unconditional random variable with the appropriate distribution.

We construct an "unconditioned" lookahead bound for each A_j as follows. Randomly choose a subset $\mathscr{U}_{s(j)} \subseteq Reach(S_{s(j)})$ in accordance with the probability distribution $\{p(B,s(j))\}$, and independently choose a duration time $D_k + X_{j,k}$ for each site $Site_k \in \mathscr{U}_{s(j)}$. $D_k + X_{j,k}$ will replace the actual corresponding duration time $D_k + Y_{j,k}$. Randomly and independently choose some value $D_{s(j)} + W_{j,s(j)}$ from $S_{s(j)}$'s duration time distribution. If $S_{s(j)}$ is a server with infinite capacity, we take $W_{j,s(j)} = \infty$. $D_{s(j)} + W_{j,s(j)}$ will replace the actual $N_j(w_n)$. Let $Z_{j,s(j)}$ be an independent exponential having the distribution of the stochastic portion of $S_{s(j)}$'s duration time. $Z_{j,s(j)}$ will replace $R_j(w_n)$; note that the residual of an $S_{s(j)}$ duration time is always as large as the residual of the duration time's stochastic portion.

The event $\mathscr{G}_j(w_n)$ gives us information that $K_j(w_n)$ is probabilistically larger than it would be if its components had their original distributions. Therefore, intuition suggests the veracity of the following bound;

Conjection 4.1

$$\lim_{n\to\infty} E[\delta(w_n) - w_n] \geq \lim_{n\to\infty} E\left[\min_{A_j \in \mathscr{A}(w_n)} \left\{Z_{j,s(j)} + \min\left\{D_{s(j)} + W_{j,s(j)}, \min_{Site_k \in \mathscr{U}_{s(j)}} \{D_k + X_{j,k}\}\right\}\right\}\right]. \tag{5}$$

Note that the expectations are not conditioned on $\mathscr{A}(w_n) = V$, and that we only require the inequality to hold in the limit of $n \to \infty$. It seems exceedingly difficult to formally establish this bound. Our analysis therefore proceeds by assuming its validity.

We continue the analysis by placing stochastic lower bounds on variables comprising the conditional (on $\mathscr{A}(w_n) = V$) expectation

$$E\left[\min_{A_j \in V} \left\{Z_{j,s(j)} + \min\left\{D_{s(j)} + W_{j,s(j)}, \min_{Site_k \in \mathscr{U}_{s(j)}} \{D_k + X_{j,k}\}\right\}\right\}\right]. \tag{6}$$

As a first step, we note that

$$\min_{Site_k \in \mathscr{U}_{s(j)}} \{D_k + X_{j,k}\} \geq \min_{Site_k \in \mathscr{U}_{s(j)}} \{X_{j,k}\} + D_{\min}. \tag{7}$$

Next we put a stochastic lower bound on $\min\{X_{j,k} | Site_k \in \mathscr{U}_{s(j)}\}$. This random variable is complicated by the fact that $\mathscr{U}_{s(j)}$ is a random set. For any *given* set $\mathscr{U}_{s(j)} = B_i$, the minimum of $|B_i|$ exponentials is itself an exponential, with rate

$\lambda_{B_i} = \sum_{Site_k \in B_i} \lambda_k$. Consequently $\min\{X_{j,k} | Site_k \in \mathscr{U}_{s(j)}\}$ is a probabilistic mixture of exponentials—with probability $p(B_i, s(j))$ it is an exponential with rate λ_{B_i}. Without loss of generality we may enumerate all subsets $B_i \subseteq Reach(S_{s(j)})$ in such a way that $\lambda_{B_i} \leq \lambda_{B_j}$ whenever $i < j$. Given this ordering, Lemma A.1 establishes that an exponential $T_{j,s(j)}$ whose rate is the "expected" rate $\psi_{s(j)} = \sum_{B_i} p(B_i, s(j))\lambda_B$ (see expression (1)) is stochastically smaller than the minimum:

$$\min_{Site_k \in \mathscr{U}_{s(j)}} \{X_{j,k}\} \geq_{st} T_{j,s(j)}.$$

Applying inequalities (7) and (13) we determine that

$$\begin{aligned}
&\min\left\{D_{s(j)} + W_{j,s(j)}, \min_{Site_k \in \mathscr{U}_{s(j)}} \{D_k + X_{j,k}\}\right\} \\
&\quad \geq \min\left\{D_{s(j)} + W_{j,s(j)}, \min_{Site_k \in \mathscr{U}_{s(j)}} \{X_{j,k}\} + D_{\min}\right\} \\
&\quad \geq_{st} \min\{D_{\min} + W_{j,s(j)}, T_{j,s(j)} + D_{\min}\} \\
&\quad = \min\{W_{j,s(j)}, T_{j,s(j)}\} + D_{\min}. \qquad (8)
\end{aligned}$$

Since $W_{j,s(j)}$ and $T_{j,s(j)}$ are both exponential, their minimum is also exponential and has rate $\gamma_{s(j)}\lambda_{s(j)} + \psi_{s(j)}$ (recall that $\gamma_{s(j)} = 0$ and thus $W_{j,s(j)} = \infty$ if $S_{s(j)}$ is a server with infinite capacity). Let $U_{j,s(j)}$ be an exponential with rate $\gamma_{s(j)}\lambda_{s(j)} + \psi_{s(j)}$. Inequality (8) holds for every $A_j \in V$; furthermore, the lookahead random variable constructed for each A_j is independent of all others. Since the addition and min operators are increasing, it follows from (13) that the expectation in (6) is bounded from below:

$$\begin{aligned}
&E\left[\min_{A_j \in V}\left\{Z_{j,s(j)} + \min\left\{D_{s(j)} + W_{j,s(j)}, \min_{Site_k \in \mathscr{U}_{s(j)}} \{D_k + X_{j,k}\}\right\}\right\}\right] \\
&\quad \geq E\left[\min_{A_j \in V}\{Z_{j,s(j)} + U_{j,s(j)}\}\right] + D_{\min}. \qquad (9)
\end{aligned}$$

We remove the conditioning on V by taking the expectation with respect to $\mathscr{A}(w_n)$,

$$\begin{aligned}
&E\left[\min_{A_j \in \mathscr{A}(w_n)}\left\{Z_{j,s(j)} + \min\left\{D_{s(j)} + W_{j,s(j)}, \min_{Site_k \in \mathscr{U}_{s(j)}} \{D_k + X_{j,k}\}\right\}\right\}\right] \\
&\quad \geq E\left[\min_{A_j \in \mathscr{A}(w_n)}\{Z_{j,s(j)} + U_{j,s(j)}\}\right] + D_{\min}.
\end{aligned}$$

If $\mathscr{A}$ has the limiting distribution of $\mathscr{A}(w_n)$ as $n \to \infty$ (supposing it exists), then

$$\begin{aligned}
&\lim_{n \to \infty} E\left[\min_{A_j \in \mathscr{A}(w_n)}\left\{Z_{j,s(j)} + \min\left\{D_{s(j)} + W_{j,s(j)}, \min_{Site_k \in \mathscr{U}_{s(j)}} \{D_k + X_{j,k}\}\right\}\right\}\right] \\
&\quad \geq E\left[\min_{A_j \in \mathscr{A}}\{Z_{j,s(j)} + U_{j,s(j)}\}\right] + D_{\min}. \qquad (10)
\end{aligned}$$

Our next task is to deal with the randomness of the set $\mathscr{A}$.

Let the collection of servers in the domain be enumerated as $V_1, V_2, \ldots$. Then let

$$\omega_i = \lim_{n \to \infty} E[\text{number of } Site_i \text{ activities occurring at time } w_n],$$

$$\rho_j = \lim_{n \to \infty} \Pr\{\text{at time } w_n \text{ an activity is occurring at } V_j\},$$

and observe that

$$\omega_i = \sum_{\text{all servers } V_j \text{ at } Site_i} \rho_j,$$

assuming that the expectations and limits exist. It is not obvious that ω_i should be identical to the equilibrium expected number of activities occurring at $Site_i$; intuitively, one expects it to be close, because the number of windows in which a given activity is found occurring is roughly proportional to the duration of the activity.

The expectation on the right-hand-side of (10) is taken with respect to a distribution of random sets of activities found occurring at a window edge. One can equivalently view it as an expectation taken with respect to a random set of servers found busy at a window edge. Inequality (9) suggests we associate two exponentials with each server V_j: Z_j and U_j (here binding j to the server rather than to the activity). There is a one-to-one correspondence between a random subset of servers, and a random subset $H \subseteq \{(Z_1, U_1), (Z_2, U_2), \ldots\}$.

Lemma A2 was developed to deal with the situation at hand. Following its statement we define

$$\begin{aligned}
\Lambda &= \sum_{j=1}^{\infty} \Pr\{(Z_j, U_j) \in H\} \lambda_{v(j)} (\gamma_{v(j)} \lambda_{v(j)} + \psi_{v(j)}) \\
&= \sum_{j=1}^{\infty} \rho_j \lambda_{v(j)} (\gamma_{v(j)} \lambda_{v(j)} + \psi_{v(j)}) \\
&= \sum_{\text{all servers } S_i} \omega_i \lambda_i (\gamma_i \lambda_i + \psi_i). \qquad (11)
\end{aligned}$$

The lemma's conclusion is that

$$E\left[\min_{(Z_j, U_j) \in H} \{Z_j + U_j\}\right] \geq \left(\frac{\pi}{2\Lambda}\right)^{1/2}.$$

The left-hand-side of this inequality is identical to the right-hand-side of (10), except for the inclusion of $D_{\min}$. Assuming the validity of Conjecture 4.1 we may conclude that

$$\lim_{n \to \infty} E[\delta(w_n) - w_n] \geq D_{\min} + \left(\frac{\pi}{2\Lambda}\right)^{1/2}. \qquad (12)$$

In order to determine the average number of events processed per window, we need to consider the rate at which events are generated by the simulation. Let $\tilde{\omega}_i$ be the equilibrium mean number of activities occurring at S_i. There are two events associated with each activity at a server with infinite capacity, begin and complete. Adding enable, there are three events associated with an

activity at a site with finite servers. We therefore define the variable ϵ_i to be 2 or 3 depending on whether server S_i has infinite capacity or note, respectively.

An activity's duration at S_i has rate $\lambda_i^{(d)} = 1/(D_i + \mu_i)$, so that the equilibrium event creation rate is $\Lambda_{Sys} = \Sigma_{\text{all servers } S_i} \epsilon_i \tilde{\omega}_i \lambda_i^{(d)}$. By flow balance, this is also the equilibrium event completion rate. We can therefore multiply this rate times the lower bound on the mean window width to bound the mean number of events processed in a window.

THEOREM 4.2. *Let*

$$\Lambda_{Sys} = \sum_{\text{all servers } S_i} \epsilon_i \tilde{\omega}_i \lambda_i^{(d)}$$

be the system event creation rate, and let

$$\Lambda = \sum_{\text{all servers } S_i} \omega_i \lambda_i (\gamma_i \lambda_i + \psi_i).$$

Then if conjecture 4.1 *is valid, the average number of events processed per window is at least*

$$\Lambda_{Sys} \left(D_{min} + \left(\frac{\pi}{2\Lambda} \right)^{1/2} \right).$$

This theorem demonstrates how an existing minimal duration time accelerates performance. Given constant $D_{\min} > 0$, the bound increases at least linearly as the total simulation event rate increases. In fact, this behavior in no way needs many of the assumptions upon which Theorem 4.2 depends. Recall inequality (4), which is derivable under the assumption that every activity require at least $D_{\min}$ time, and that completion messages are pre-sent. If every window has width at least $D_{\min}$, then we have the following result

THEOREM 4.3. *Let*

$$\Lambda_{Sys} = \sum_{\text{all servers } S_i} \epsilon_i \tilde{\omega}_i \lambda_i^{(d)}$$

be the system event creation rate. If the duration of every activity is at least as large as D_{min} and if every completion message is pre-sent, then the average number of events processed per window is at least

$$\Lambda_{Sys} D_{min}.$$

However, much of the analysis in this paper was developed specifically to demonstrate that even when $D_{\min} = 0$, good performance is also possible, as we now demonstrate.

The value of Λ is defined in terms of ω_i. We have no immediate cause for believing that $\omega_i = \tilde{\omega}_i$; nor is it clear that the two quantities should be widely different. It seems reasonable then to take $\omega_i \approx \tilde{\omega}_i$ as a first approximation. Doing so permits us to analytically estimate Λ in some simple cases, and quantify the bound given by Theorem 4.2.

As pointed out by Wagner and Lazowska [32], interconnection topology plays an important role in determining the performance one achieves with a queuing system. Network bottlenecks limit the volume of simulation activity. This is reflected in Theorems 4.2 and 4.3. For example, in a network where each site

has one server, ω_i is approximately the server utilization. A bottleneck server will have a very high utilization while those at other servers are comparatively low. After a point, adding jobs to the network does not appreciably increase the sum of server utilizations, hence the overall event rate does not appreciably increase. For the same reason, simulated queuing systems are constrained even if the throughput at each server is equal. The overall system event rate is maximized when all server utilizations are one. After a point, to increase simulation activity one needs to increase the size of the network.

We can approximate the bound in Theorem 4.2 in some simple cases. Consider a model where objects move throughout the domain. An object resides at a site for a fixed time $D_{\min}$ plus an exponential time with mean $1/\lambda$, and then moves to any other site, chosen uniformly at random and independently of any other object. Equilibrium flow balance equations are easily solved in this situation. Working through the details with K objects and $D_{\min} = 0$, one discovers that at least $\sqrt{\pi K/2}$ events are processed per window, on average. A relevant point is that the intersite communication topology is that of a fully connected graph. Such topologies are generally taken to be extremely taxing on conservative synchronization methods, because the "next" event at a site can come from anywhere. Nevertheless, a significant amount of work is performed for each window, at least when K is large. Figure 2 plots the analytically bounded and empirical measured average number of events processed per window, as a function of $\log_2 K$. The empirical measurements represent the sample mean of ten long simulation runs. There was very little variance between these runs. Figure 2 shows that if thousands of objects are in the model, hundreds of events are processed each window. Since parallel processing techniques are used primarily when serial processing times are too slow (or memories are too small), we see that this result applies directly to situations of practical interest—large simulation models on medium scale parallel architectures.

Performance is greatly enhanced when $D_{\min} > 0$. Figure 3 plots measurements of the number of events per window for small models, having only 256 and 1024 objects. The same measurement methodology as was described for Figure 2 is used here. The analytic bound is not displayed, being indistinguishable from the measured performance when plotted on the graph. $D_{\min}$ is varied between 0 and $\mu = 1/\lambda$, so that $D_{\min}/\mu$ varies between 0 and 1. We see that if a model has minimal duration times we can expect many more events per window than if not. Note that the protocol does not need to know $D_{\min}$, as it is already part of the presampled duration times. Dramatic performance improvement as one's ability to "look ahead" increases has also been observed by Fujimoto [6].

Our confidence in the conclusions of Theorem 4.2 is increased by the fact that the approximated lower bound did uniformly fall below measured performance. Similar results have been observed when comparing the measured and bounded performance on less homogeneous simulation models.

5. *The Cost of Conservative Synchronization*

Next we consider the overheads involved in implementing this conservative protocol. First we identify conditions under which the average number of events processed per window will grow without bound as the system event

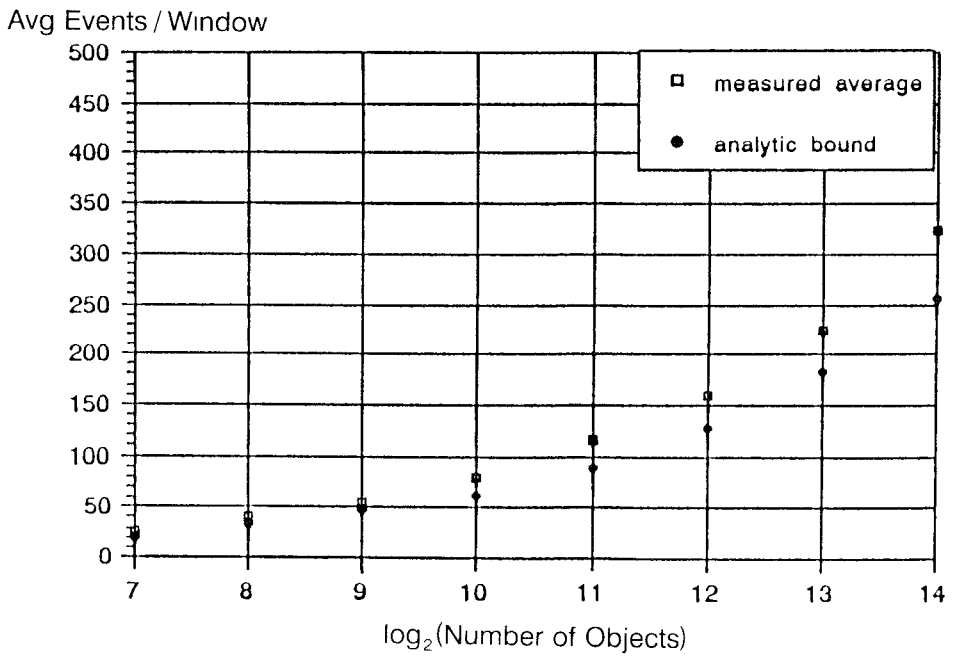

FIG. 2. Average events per window, as function of number of objects. $D_{\min} = 0$; durations are homogeneous exponentials; no queuing; routing is uniformly random.

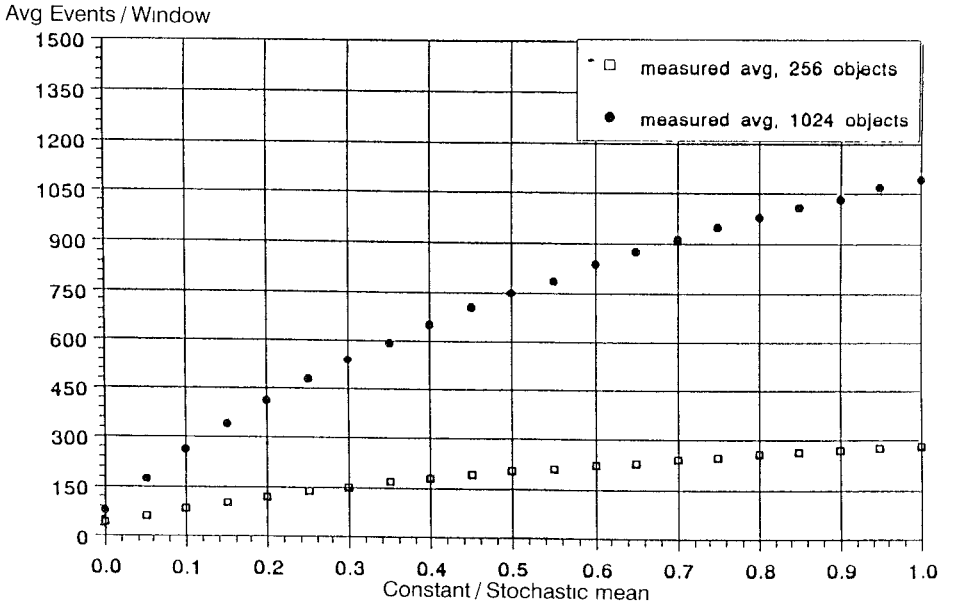

FIG. 3. Average events per window when $D_{\min} > 0$. Performance plotted as function of $D_{\min}/\mu$, for 256 and 1024 objects.

creation rate grows without bound. Then we show that, as the number of events processed per window grows, our method's per event overhead due to synchronization, processor idle time, lookahead calculation, and event list manipulation becomes (within a constant factor of average) the per-event overhead of performing the simulation serially.

One way of increasing the system event creation rate Λ_{Sys} is to increase the "size" of the model. For example, we increase the size of the moving objects simulation described earlier by increasing the number of objects in the domain. We may also increase the number of sites, although in this case it is not necessary. Theorems 4.2 and 4.3 show how the average number of events processed within each window may increase as Λ_{Sys} increases: least $\Lambda_{Sys} D_{min}$ events are processed within each window on average. It is also possible for the average number of events to increase without bound as Λ_{Sys} increases even when $D_{min} = 0$. For example, suppose there is a value α such that as the size of the simulation model is increased the following bound is true for all servers S_i:

$$\frac{\omega_i \lambda_i (\lambda_i + \psi_i)}{\tilde{\omega}_i \lambda_i^{(d)}} \leq \alpha.$$

This condition is a formal statement that as the model size grows ψ_i cannot get too large relative to λ_i, and that any difference between ω_i and $\tilde{\omega}_i$ does not get too large. The first condition will be satisfied if there exist $\lambda_{\max}$ and $R_{\max}$ such that as the model size grows, $\lambda_i \leq \lambda_{\max}$ and $|Reach(S_i)| \leq R_{\max}$, for all i. The second condition ought to be satisfied if our intuition that $\omega_i \approx \tilde{\omega}_i$ is correct. If the bound above holds, then, as the model size grows, the inequality $\Lambda \leq \alpha \Lambda_{Sys}$ will always hold. It follows that at least $\sqrt{\pi \Lambda_{Sys}/(2\alpha)}$ events are processed within each window on average, a number that grows without bound as Λ_{Sys} grows without bound.

As a point of comparison, we assume that a serial implementation uses the best-known event-list management algorithm. If there are T total events in the system on average, we let $f(T)$ be the average complexity per operation of an optimized serial event-list algorithm. For example, there is some evidence that a "calendar-queue" implementation has an average $O(1)$ complexity (i.e., $f(T) = 1$) on the hold model [3]. A number of other event list algorithms exhibiting $\Omega(\log T)$ average complexity are also commonly used [10]. We assume that the serial event-list algorithm permits the deletion of a nonminimal element without affecting the overall average complexity. This assumption is satisfied by the calendar queue implementation.

We make the reasonable assumption that as the simulation model size is increased, T grows at least as rapidly as S, that is, $T = \Omega(S)$.

Now consider a parallel simulation that uses our protocol. The requirement that completion notices be pre-sent may increase the average total number of events in event lists at a time. This represents a factor of two increase, at most. In the complexity analysis to follow we need not explicitly concern ourselves with this constant factor increase.

One overhead suffered in the parallel simulation is the cost of computing $\delta_i(w_n)$ for each server S_i. This value may change only when an event is inserted or deleted at S_i's site. A site with finite servers can recompute this value with $O(1)$ cost whenever its event list changes. A site with an infinite capacity server

can organize the completion times of its pending activities in a priority queue using whatever event list algorithm is employed by the optimized serial implementation. The minimum value in the priority queue defines δ_t. The priority queue is modified only when the site's event list is modified, at cost $O(f(T))$. A processor can organize the δ_t values from each of its servers into a priority queue, enabling it to determine the minimum on-processor δ_t value at least as quickly as the optimized serial implementation finds its minimal element. Maintenance of this priority queue costs $O(f(S))$ on average for each processed event. Once each processor has determined its locally minimum δ_t value, all processors may cooperatively compute the global minimum in C_{Synch} time. Note that our assumption that P is fixed permits us to ascribe a worst-case constant cost to this operation.

Another overhead is processor idle time. The protocol is punctuated with global synchronizations, between which the processors execute in parallel. A processor with little workload will spend a long time waiting for more heavily loaded processors to reach the synchronization barrier. Suppose there are W events to process in a window. For the purposes of analysis, assume that each event may be mapped to any processor, with equal probability.[1] Then the number of events assigned to a processor is a binomial $B(W, 1/P)$ random variable. The collection of workload random variables are not independent however, as we know they must sum to W. However, it is not difficult to construct a coupling [27] argument to show that the expected maximum workload of this system must be smaller than the expected maximum workload of a system where each processor has an independent $B(W, 1/P)$ workload. The binomial distribution has an increasing hazard rate function [27, p. 280]; it is therefore stochastically less variable than an exponential with the same mean [28, p. 273], and hence the expected maximum of P independent exponential random variables with mean W/P is at least as large as the expected maximum of P independent $B(W, 1/P)$ random variables. The expected maximum of the exponentials is approximately $(W/P)\ln(W/P)$. Assuming each event takes the same amount of time to process, the average fraction of time a processor is left idle is no greater than

$$1 - \frac{(W/P)}{(W/P)\ln P} = 1 - \frac{1}{\ln P}.$$

This implies that the average overhead cost per event due to processor idleness is $O(1)$. This analysis is actually quite pessimistic. Much better load balance can be achieved through use of mapping techniques such as scatter decomposition [24]. Also, the bound above is insensitive to increasing volume of workload, whereas in practice the proportion of idle time tends to decrease as the volume of workload increases.

The complexity of the average per-event overhead due to event list manipulation, lookahead calculations, processor idle time, and global synchronization is

$$\frac{W(O(f(T)) + O(f(S)) + O(1)) + C_{Synch}}{W} = O(f(T)).$$

[1] This can't rigorously be true, since events at the same site are evaluated on the same processor. It is a reasonable approximation when W is large compared with P.

Relative to the serial simulation, performance must then be within a constant factor of optimal, at least if inter-*LP* communication costs are ignored (we have already accounted for the communication costs that are specific to the protocol). Inter-*LP* communication costs are dependent on the simulation model and its mapping, and are dependent on the architecture. It is possible for communication costs to overwhelm performance, even if our protocol finds a great deal of parallel workload. However, these costs are inherent to the model, and would be suffered under any synchronization protocol.

6. *Empirical Results*

We used the protocol analyzed in this paper in a parallel discrete-event simulation testbed implemented on the Intel iPSC/2 distributed memory multiprocessor [2]. The testbed, YAWNS (Yet Another Windowing Network Simulator) [23], is designed to permit rapid development of simulation models, by providing a framework within which all synchronization and interprocessor communication activity is automated, and hidden from the user. YAWNS uses a computational paradigm where the simulation model is decomposed into communicating *Logical Processes* (*LPs*). *LP*'s interact by passing messages. A site in our analytic model plays the role of an *LP*.

The simulation modeler must provide the testbed with three routines for each *LP* (the *LP*'s may share these routines). One routine processes messages, typically inserting an event into the *LP*'s event list as a result. This routine is responsible for choosing a duration time for the enabled activity. Another routine processes events. Messages to other *LP*s may be generated as a result of calling this routine; these messages correspond to the completion messages described in the analytic model. The third routine is called to obtain the lookahead value required of an *LP*. YAWNS demands that the simulation modeler know about the protocol only to the extent that inter-*LP* messages are pre-sent, and an *LP* must be able to determine a lower bound on the time of the next message it sends.

It is always important to use the best possible event list algorithm for an *LP*. YAWNS provides a linearly linked list algorithm for use when the number of events in an *LP*'s list is small, and a splay-tree algorithm for large lists.

We report on the performance achieved by four diverse applications: the moving objects simulation described earlier, a logic network, the game of Life, and a timed Petri net model. All measurements reported are taken from a thirty-two processor machine. Each simulation model was run long enough to generate several millions of events. The execution time was typically a minute or two, once the problem was loaded and running. Much longer runs were also performed, but no appreciable difference in performance statistics were observed.

The measured performance supports our analysis, and actually becomes quite good on large problems. The metric we use to gauge performance is average processor utilization, measured as the fraction of time a processor spends doing work that would be performed in a serial implementation of the simulation, using the same paradigm. Time spent in computing lookahead, synchronization, interprocessor communication, and idle time are explicitly counted as overhead, and do not appear in the utilization figure. One can translate such efficiencies into "speedup" figures by multiplying by the number

of processors used, provided the resulting numbers are properly interpreted. The speedup so computed is relative to a serial version that uses the same paradigm (and code) of communicating *LP*s as is used in the parallel version. This is not an unreasonable paradigm for a general purpose serial simulation system, but is not likely to be the paradigm of choice for a serial version that is highly optimized for the given application. In our experience (and depending on the application), the communicating *LP* paradigm is a factor of 1.5 to 2 times slower than an optimized serial version. The usual comparison of serial running time to parallel running time is impossible to directly obtain as the largest models we simulate are too large for a single processor's memory. We shall see that on the largest problems the average processor utilization ranges from 60%–90%.

Each model is quite homogeneous, thereby trivializing the problem of assigning *LP*s to processors. Our performance measurements should be viewed in that light. The load balancing of parallel discrete-event simulations is a problem we are very much interested in, but have not yet seriously addressed in YAWNS.

6.1. MOVING OBJECTS. This problem is identical to that described elsewhere in this paper, save that the sites are connected in a hypercube topology. Each object resides at a site for a time constructed by adding 0.25 to an exponential with mean 1, and then moves randomly to a site connected with its present one. This model can also be viewed as a queuing network of infinite servers. This model does not capture the interaction between objects found in some physical models of colliding particles, but does resemble physical models where the effects of particle interactions are not explicitly computed, but are modeled instead with stochastic movement.

In each model, there are exactly as many objects as there are sites. We increase the size of the problem by simultaneously increasing the number of objects and the number of sites. We may therefore describe the size of the system by the dimension of the underlying hypercube. Pre-sent completion times and lookahead values are computed exactly as described earlier for sites with an infinite capacity server. Average processor utilization ρ as a function of hypercube dimension is given below. Many simulation runs were performed, the variance in the timing numbers is quite small.

Dim	8	9	10	11	12	13	14
ρ	21%	28%	34%	46%	54%	60%	62%

6.2. LOGIC NETWORK. To ensure that we simulated networks with high concurrency, we constructed "random" logic networks having the topology of a butterfly interconnection network. The last stage wraps around to feed the first. Each gate was randomly assigned to be an AND, OR, or XOR function and was given a randomly chosen gate delay time of 1,2, or 3 time units. Each gate was modeled as an *LP*. These experiments assume constant time gate delays so that the eventual output of a gate whose inputs have changed can be computed at the time the inputs change, and gate state changes can be pre-sent. A gate is like a site with an infinite capacity server; its lookahead is computed to be the minimum time of the next input change plus the gate delay. The size of network can be described by the dimension of a column of

gates. For example, a network of dimension 6 has 6 columns, each composed of 2^6 gates. Observed performance is given below.

Dim	5	6	7	8	9	10	11
ρ	24%	32%	43%	52%	59%	66%	70%

More realistic logic simulations permit variable gate delays, depending on whether the state transition is $0 \rightarrow 1$ or $1 \rightarrow 0$. When our protocol is interpreted in the obvious way it will not deal with these situations. For example, suppose a $0 \rightarrow 1$ transition takes 3 time units, a $1 \rightarrow 0$ transition takes 1.5 units, and a $0 \rightarrow 1$ transition is triggered at time 10. Nominally, we expect the gate's new state to be 1 at time 13. However, the gate's inputs may change again at time 10.5 in such a way that the gate's state should be fixed at zero after time 12. It would therefore be erroneous to send a completion message at the time 10 signaling a $0 \rightarrow 1$ change at time 13. However, a simple reinterpretation of the protocol deals with the problem.

Define a $0 \rightarrow 1$ transition as being composed of two sequential activities, both with duration 1.5 units. The first activity serves to see if the tentative transition is interrupted; the second activity completes the transition. Returning to the example above, at time 10 the gate sends a "completion" message to itself with time-stamp 11.5: it knows that any interruption to the tentative transition must occur by time 11.5. By the time the window containing 11.5 is defined, the gate will have received notification that its inputs changed again at time 10.5. The second activity will not then be initiated. If the tentative transition had not been interrupted, then at time 11.5 the gate can send a state-change message with time-stamp 13 to all gates it feeds.

In general, the protocol can be interpreted to deal with preemption so long as there is a minimal delay $D_{\min}$ between the initiation of a preemptive activity and the time when the preemption affects the model. If activities are properly defined, one can detect such preemption before executing an erroneous event.

6.3. CONWAY'S GAME OF LIFE. Initial random configurations were chosen so that the probability of a cell being alive is at step 0 is 0.2. Each cell is modeled as an *LP*. A cell is evaluated at step n only if one of its neighbors (or itself) is alive at step $n - 1$. It is straightforward to pre-send "new state" messages; lookahead consists of one step time. The problem size is increased by increasing the size of the board. Again, we can easily describe problem size in terms of dimension. A $2^j \times 2^j$ board will be said to have dimension j.

Dim	3	4	5	6	7	8
ρ	12%	16%	35%	54%	69%	77%

Larger problems than a 256×256 board will often exhaust the available dynamic memory in some processor, after some period of execution. This points out one of consequences of internally buffering all messages until the window's workload is completed.

6.4. TIMED PETRI NETS. Consider a timed Petri net model of a multiprocessor system organized with a mesh communication topology. The net models a system where a processor iteratively receives a message from each of its NEWS (North, East, West, South) neighbors, performs a computation, and sends a

result to each NEWS neighbor. The net models a flow control policy that prevents a processor from sending a message to a neighbor until the last message it sent to that neighbor is consumed. An *LP* consists of the network for one processor, a network containing approximately thirty places and ten transitions. Nearly all transitions have a unit-time delay associated with them. Transitions modeling the processor execution time have 200 units of delay.

This Petri net model does not satisfy exactly the assumptions we have made concerning simulation model behavior. The main difference is that a token arriving to an *LP* does not trigger a single *LP* activity with a single duration time. The response of the *LP* is liable to be much more complex. Nevertheless, the basic synchronization protocol works. Tokens from an enabled transition are always pre-sent (regardless of whether they are sent to places within the *LP*); to compute lookahead, an *LP* adds the minimum delay among all transitions that send tokens to other *LP*s to the least-time token arrival event in the *LP*'s event list.

The grid size for the simulated system can be described in terms of dimension in the same way as was the Game of Life.

Dim	3	4	5	6
ρ	35%	62%	84%	94%

The comparatively better performance of this problem can be attributed to its better ratio of computation costs to *LP*-message costs.

7. *Conclusions*

We have analyzed a simple conservative synchronization protocol for parallel discrete-event simulation. The protocol presumes that one can pre-sample activity duration times (or bound those times from below), that the immediate effects of simulation model state changes are very local, and that all queuing disciplines are non-preemptive. The protocol essentially slides a window across simulation time; the window is defined so that processors can evaluate all their window events in parallel. We construct a lower bound on the average number of events processed per window. One bound applies in very general situations, and requires only that there be a non-zero lower bound on the time any activity requires. Another bound is sharper (but approximate), and permits arbitrary small positive activity durations. The bounds depend on the topology and activity rates of the heterogeneous simulation domain. The performance analysis shows that a great deal of workload can be performed in parallel, if there is a great deal of concurrent activity in the simulation model. Non-zero minimal activity durations are shown to greatly improve performance. We show that the asymptotic time complexity of the average total overhead (synchronization, lookahead calculations, processor idle time, event list manipulation) per event is that of an optimized serial simulation. Assuming that the complexity of the communication cost per event is no greater than the overhead of an event in a serial implementation, the protocol's performance is within a constant factor of optimal. The region of problems where the method does well is precisely the region where parallel processing is most effectively applied—problems too large to run serially. The method is verified by implementation on a distributed

memory multiprocessor. Good performance is observed on a variety of problems.

Appendix A

In this appendix, we describe the tools used in our analysis, and develop some key results.

A1. STOCHASTIC DOMINANCE. Our analysis relies on the theory of stochastic dominance. The definitions and results we cite are taken from Ross [27, chap. 8].

Random variable X is said to be *stochastically larger* than random variable Y if for all t

$$\Pr\{X > t\} \geq \Pr\{Y > t\}.$$

We then write $X \geq_{st} Y$, or $Y \leq_{st} X$. An equivalent definition is that

$$E[g(X)] \geq E[g(Y)] \quad \text{for all increasing functions } g.$$

In particular, $E[X] \geq E[Y]$. If $X_1, \ldots, X_n$ are independent random variables and $Y_1, \ldots, Y_n$ are independent random variables such that $X_i \geq_{st} Y_i$ for all i, then for all increasing functions g,

$$g(X_1, \ldots, X_n) \geq_{st} g(Y_1, \ldots, Y_n).$$

A2. HAZARD RATE FUNCTIONS. If X is a nonnegative continuous random variable, it has a *hazard rate function*, also known as a *failure rate function*. Let $f(t)$ be X's density function, and let $\bar{F}(t) = \Pr\{X > t\}$. Then X's hazard rate function is defined to be

$$\lambda(t) = \frac{f(t)}{\bar{F}(t)}.$$

If X is exponential, then $\lambda(t)$ is identically the exponential's rate parameter.

We rely on the following results concerning hazard rate functions.

—If $\lambda_X(t)$ and $\lambda_Y(t)$ are hazard rate functions for X and Y, and $\lambda_X(t) \leq \lambda_Y(t)$ for all t, then $X \geq_{st} Y$.

—If $X_1, \ldots, X_n$ are independent random variables with hazard rate functions $\lambda_1(t), \ldots, \lambda_n(t)$, then the hazard rate function for $\min\{X_1, \ldots, X_n\}$ is simply $\sum_{i=1}^{n} \lambda_i(t)$.

—If X has hazard rate function $\lambda(t)$, then for any t and s, $s \leq t$,

$$\Pr\{X > t | X > s\} = \exp\left\{-\int_s^t \lambda(u)\, du\right\}.$$

This also shows (taking $s = 0$) that the hazard rate function uniquely defines a distribution.

A3. IMPORTANT BOUNDS. We now establish some important bounds used in this paper.

Random variables constructed by randomly choosing one of a set of random variables are called *mixtures*. The following lemma bounds the hazard rate of a certain class of mixtures.

LEMMA A1. *Let $X_1, X_2, \ldots,$ be independent random variables with hazard rate functions $\lambda_1(t), \lambda_2(t), \ldots$ and suppose that these functions are ordered:*

$$\lambda_i(t) \leq \lambda_{i+1}(t) \qquad \text{for all } i = 1, 2, \ldots, \text{ and all } t \geq 0.$$

Let $p_1, p_2, \ldots$ be probabilities such that $\sum_{i=1}^{\infty} p_i = 1$, and consider the random variable M constructed by randomly selecting some X_i, with probability p_i. Let $\lambda_M(t)$ be M's hazard rate function. Then for all $t \geq 0$

$$\lambda_M(t) \leq \sum_{i=1}^{\infty} p_i \lambda_i(t).$$

PROOF.[2] Let $f_i(t)$ and $F_i(t)$ be the density and cumulative distribution functions for X_i. Then $\lambda_i(t) = f_i(t)/\bar{F}_i(t)$, and

$$\lambda_M(t) = \frac{\sum_{i=1}^{\infty} p_i f_i(t)}{\sum_{i=1}^{\infty} p_i \bar{F}_i(t)}.$$

The desired conclusion will follow if

$$\sum_{i=1}^{\infty} p_i f_i(t) \leq \left(\sum_{i=1}^{\infty} p_i \bar{F}_i(t) \right) \left(\sum_{i=1}^{\infty} p_i \lambda_i(t) \right)$$

for all t. Let $Y = i$ with probability p_i and let $h(Y, t) = \lambda_Y(t)$ and $g(Y, t) = -\bar{F}_Y(t)$. Then for every fixed t, h and g are increasing in Y. Application of Proposition 7.1.5 of [27, p. 227] yields

$$E[h(Y, t)]E[g(Y, t)] \leq E[h(Y, t)g(Y, t)],$$

or equivalently,

$$\sum_{i=1}^{\infty} p_i f_i(t) \leq \left(\sum_{i=1}^{\infty} p_i \bar{F}_i(t) \right) \left(\sum_{i=1}^{\infty} p_i \lambda_i(t) \right).$$

As this holds for every $t \geq 0$, the lemma's conclusion follows. □

We now develop a lower bound on the expected minimum of a random number of variables, each variable being the sum of two exponentials.

LEMMA A2. *Let $\mathcal{S} = \{(Z_1, U_1), (Z_2, U_2), \ldots,\}$ be a countable set where Z_i is exponential with rate λ_i, and U_i is exponential with rate ψ_i. Let all these random variables be independent. Let $B_1, B_2, \ldots$ be the set of finite subsets of $\mathcal{S}$. Let B be a random set constructed by choosing B_i with probability p_i. Let*

$$\Lambda = \sum_{i=1}^{\infty} Pr\{(Z_i, U_i) \in B\} \lambda_i \psi_i.$$

Then

$$E\left[\min_{(Z_i, U_i) \in B} \{Z_i + U_i\} \right] \geq \left(\frac{\pi}{2\Lambda} \right)^{1/2}.$$

[2] This elegant proof was suggested by an anonymous referee, replacing a far more complicated proof of our own.

PROOF. Consider the hazard rate function $\gamma_t(t)$ for $Z_t + U_t$. This random variable is the lifetime of a serial two-stage system where the first stage lasts for time Z_t, and the second lasts for time U_t. $\gamma_t(t)$ is the instantaneous probability density associated with the system dying at time t, given that it has survived up to time t. Now if $Z_t > t$, the system cannot fail at t, whence $\gamma_t(t) = 0$. If $Z_t \leq t$, then the hazard rate is simply that of U_t: ψ_t. Note that this observation relies on the memoryless property of the exponential. We may therefore write

$$\begin{aligned}\gamma_t(t) &= (1 - \Pr\{Z_t > t | Z_t + U_t > t\})\psi_t \\ &\leq (1 - \Pr\{Z_t > t\})\psi_t \\ &= (1 - \exp\{-t\lambda_t\})\psi_t.\end{aligned}$$

One can show that the left-hand-side of this inequality is equivalent to the more usual (and complicated) derivation of the hazard rate function for the sum of two exponentials [31 p. 126]. The function on the right-hand-side is concave in t, and is hence dominated everywhere by the line tangent to it at $t = 0$: $\tau_t(t) = t\lambda_t\psi_t$. A random variable V_t with hazard rate function $\tau_t(t)$ satisfies $V_t \leq_{st} Z_t + U_t$.

Let B_j be any finite subset of $\mathscr{S}$. By [13] and the observations above we may conclude that

$$E\left[\min_{(Z_t, U_t) \in B_j} \{Z_t + U_t\}\right] \geq E\left[\min_{(Z_t, U_t) \in B_j} \{V_t\}\right].$$

We now focus on the right-hand-side of this inequality. The hazard rate function for $M_j = \min\{V_i | (Z_t, U_t) \in B_j\}$ is simply

$$\lambda_{B_j}(t) = t \cdot \left(\sum_{(Z_t, U_t) \in B_j} \lambda_t \psi_t\right).$$

Without loss of generality, we may enumerate the finite subsets of $\mathscr{S}$ in such a way that if $i < j$, then $\lambda_{B_i}(t) \leq \lambda_{B_j}(t)$ for all t. Let M be a mixture of $\{M_1, M_2, \ldots\}$, where M_j is chosen with probability p_t; let $\lambda_M(t)$ be M's hazard rate function. By Lemma A1 we can bound $\lambda_M(t)$ from above by $\lambda_Y(t)$, defined by

$$\begin{aligned}\lambda_M(t) &\leq \sum_{t=1}^{\infty} p_t \lambda_{B_t} \\ &= \sum_{t=1}^{\infty} \Pr\{(Z_t, U_t) \in B\} t\lambda_t\psi_t = \lambda_Y(t).\end{aligned}$$

Let Y be a random variable with hazard rate function $\lambda_Y(t)$. Using the correspondence between hazard rate functions and probability distributions

(see sect. A2), we have

$$\begin{aligned}\Pr\left\{\min_{(Z_t,U_t)\in B}\{V_t\} > t\right\} &\geq \Pr\{Y > t\}\\ &= \exp\left\{-\int_0^t \lambda_Y(s)\,ds\right\}\\ &= \exp\left\{-\sum_{t=1}^{\infty}\Pr\frac{\{(Z_t,U_t)\in B\}t^2\lambda_t\psi_t}{2}\right\}\\ &= \exp\left\{-\frac{\Lambda t^2}{2}\right\}.\end{aligned}$$

Now

$$\begin{aligned}E\left[\min_{(Z_t,U_t)\in b}\{\dot{Z}_t + U_t\}\right] &= \int_0^{\infty}\Pr\{\min\{Z_t + U_t|(Z_t,U_t)\in B\} > t\}\,dt\\ &\geq \int_0^{\infty}\exp\left\{-\frac{\Lambda t^2}{2}\right\}\,dt\\ &= \left(\frac{1}{\sqrt{\Lambda}}\right)\int_0^{\infty}\exp\left\{\frac{-s^2}{2}\right\}\,ds \qquad \text{by defining } s = t\sqrt{\Lambda}\\ &= \left(\frac{\pi}{2\Lambda}\right)^{1/2}.\end{aligned}$$

□

REFERENCES

1. AYANI, R. A parallel simulation scheme based on distances between objects. In *Distributed Simulation 1989*, SCS Simulation Series. Society for Computer Simulation, San Diego, Calif., 1989, pp. 113–118.
2. BOMANS, L., AND ROOSE, D. Benchmarking the iPSC/2 hypercube multiprocessor. *Concurrency: Practice and Experience 1*, 1 (Sept. 1989), 3–18.
3. BROWN, R. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Commun. ACM 31*, 10 (Oct. 1988), 1220–1227.
4. CHANDY, K., AND SHERMAN, R. The conditional event approach to distributed simulation. In *Distributed Simulation 1989*, SCS Simulation Series. Society for Computer Simulation, San Diego, Calif., 1989, pp. 93–99.
5. CHANDY, K. M. AND MISRA, J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Softw. Eng. 5*, 5 (Sept. 1979), 440–452.
6. FUJIMOTO, R. M. Performance measurements of distributed simulation strategies. In *Distributed Simulation 1988*, vol. 19, SCS Simulation Series. Society for Computer Simulation, San Diego, Calif., 1988, pp. 14–20.
7. FUJIMOTO, R. M. Parallel discrete event simulation. *Commun. ACM 33*, 10 (Oct. 1990), 30–53.
8. GUPTA, A., AKYLIDIZ, I., AND FUJIMOTO, R. Performance analysis of Time Warp with homogeneous processors and exponential task times. In *Proceedings of the 1991 SIGMETRICS Conference* (San Diego, Calif., May 21–24). ACM, New York, 1991, pp. 101–110.
9. JEFFERSON, D. R. Virtual time. *ACM Trans. Prog. Lang. Syst. 7*, 3 (July 1985), 404–425.
10. JONES, D. W. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM 29*, 4 (Apr. 1986), 300–311.
11. LAVENBERG, S., MUNTZ, R., AND SAMADI, B. Performance analysis of a rollback method for distributed simulation. In *Performance '83*. Elsevier Science Pub. B. V. (North Holland), Amsterdam, The Netherlands, 1983, pp. 117–132.

12. LIN, Y.-B., AND LAZOWSKA, E. Exploiting lookahead in parallel simulation. *IEEE Trans. para. Dist. Syst. 1*, 4 (Oct. 1990), 457–469.
13. LIN, Y.-B., AND LAZOWSKA, E. Optimality considerations for "Time Warp" parallel simulation. In *Distributed Simulation 1990*, SCS Simulation Series. Society for Computer Simulation, San Diego, Calif., 1990, pp. 29–34.
14. LIPTON, R., AND MIZELL, D. Time Warp vs. Chandy-Misra: A worst-case comparison. In *Distributed Simulation* 1990, vol. 22, SCS Simulation Series. Society for Computer Simulation, San Diego, Calif., 1990, pp. 137–143.
15. LUBACHEVSKY, B. D. Efficient distributed event-driven simulations of multiple-loop networks. *Commun. ACM 32*, 1 (Jan. 1989), 111–123.
16. LUBACHEVSKY, B. Scalability of the bounded lag distributed discrete-event simulation. In *Distributed Simulation 1989*, SCS Simulation Series. Society for Computer Simulation, San Diego, Calif., 1989, pp. 100–107.
17. LUBACHEVSKY, B., SHWARTZ, A., AND WEISS, A. Rollback sometimes works... if filtered. In *Proceedings of the 1989 Winter Simulation Conference* (Washington, D.C.). ACM, New York, 1989, pp. 630–639.
18. MADASETTI, V., WALRAND, J., AND MESSERSCHMITT, D. Synchronization in message-passing computers: Models, algorithms, and analysis. In *Distributed Simulation 1990*, vol. 22, SCS Simulation Series. Society for Computer Simulation, San Diego, Calif., 1990, pp. 35–48.
19. MISRA, J. Distributed discrete-event simulation. *Comput. Surv. 18*, 1 (Mar. 1986), 39–60.
20. MITRA, D., AND MITRANI, I. Analysis and optimum performance of two message-passing parallel processors synchronized by rollback. In *Performance '84*. Elsevier Science Pub. B. V. (North Holland), Amsterdam, The Netherlands, 1984, pp. 35–50.
21. NICOL, D. Parallel discrete-event simulation of FCFS stochastic queuing networks. *SIGPLAN Notices 23*, 9 (Sept. 1988), 124–137.
22. NICOL, D. Performance bounds on parallel self-initiating discrete event simulations. *ACM Trans. Model. Comput. Sim. 1*, 1 (1991), 24–50.
23. NICOL, D., MICHEAL, C., AND INOUYE, P. Efficient aggregation of multiple LP's in distributed memory parallel simulations. In *Proceedings of the 1989 Winter Simulation Conference* (Washington, D. C., Dec.). ACM, New York, 1989, pp. 680–685.
24. NICOL, D., AND SALTZ, J. An analysis of scatter decomposition. *IEEE Trans. Comput. 39*, 11 (Nov. 1990), 1337–1345.
25. PEACOCK, J. K., MANNING, E., AND WONG, J. W. Synchronization of distributed simulation using broadcast algorithms. *Comput. Netw. 4* (1980), 3–10.
26. REYNOLDS, P. JR. A shared resource algorithm for distributed simulation. In *Proceedings of the 9th Annual International Computer Architecture Conference* (Austin, Texas, Apr.). ACM, New York, 1982, pp. 259–266.
27. ROSS, H. *Stochastic Processes*. Wiley, New York, 1983.
28. SARGENT, R. Event graph modelling for simulation with application to flexible manufacturing systems. *Manage. Sci. 34*, 10 (Oct. 1988), 1231–1251.
29. SCHRUBEN, L. Simulation modeling with event graphs. *Commun. ACM 26*, 11 (Nov. 1983), 957–963.
30. SOKOL, L., BRISCOE, D., AND WIELAND, A. MTW: A strategy for scheduling discrete simulation events for concurrent execution. In *Distributed Simulation 1988*. SCS Simulation Series. Society for Computer Simulation, San Diego, Calif., 1988, pp. 34–42.
31. TRIVEDI, K. *Probability and Statistics, with Reliability, Queuing, and Computer Science Applications*. Prentice-Hall, Englewood Cliffs, N.J., 1982.
32. WAGNER, D., AND LAZOWSKA, E. Parallel simulation of queuing networks: Limitations and potentials. In *Proceedings of the* 1989 *SIGMETRICS Conference* (Berkeley, Calif., May 23–26). ACM, New York, 1989, pp. 146–155.

RECEIVED JULY 1989; REVISED AUGUST 1991; ACCEPTED AUGUST 1991