# Sluggish Calendar Queues for Network Simulation

Guanhua Yan and Stephan Eidenbenz*
Discrete Simulation Sciences (CCS-5)
Los Alamos National Laboratory
{ghyan, eidenben}@lanl.gov

## Abstract

*Discrete event simulation is an indispensable tool to understand the dynamics of communication networks and evaluate their performance. As the scale and complexity of these networks increases, simulation itself becomes a computationally prohibitive undertaking. Among all possible solutions, improving the performance of event manipulation operations is an important one. In this paper, we discover that in network simulation events are often inserted into the simulation kernel in their timestamp order. Based on this observation, we make some simple modifications on the conventional calendar queue. Experiments show that the new data structure can achieve two orders of execution speedup against the conventional calendar queue in some wireline network simulation and in wireless network simulation, the speedup scales well with the network size.*

## 1 Introduction

As both scale and complexity of communication networks grew dramatically over the last several decades, evaluating their performance has become an increasingly challenging problem. Discrete event simulation has survived as an indispensable tool in understanding and optimizing the operation of these networks, because alternative approaches, particularly analytical solutions and real testbeds, either suffer from lack of tractability or require significant investment.

Discrete event simulation of large-scale communication networks is extremely computation-intensive [11]. Achieving efficient large-scale network simulation requires optimized solutions in multiple dimensions, such as parallel simulation and model abstraction [17]. In the whole solution space, efficient event management algorithms play an important role in improving network simulation performance. As large-scale network simulation inevitably generates a tremendous number of simulation events, it is not surprising that its performance is affected by how efficiently these events are managed in the simulation kernel. Experiments reveal that more than 30% of the computation time can come from event manipulation operations [3].

*Los Alamos National Laboratory Publication No. LA-UR-06-3848

Given the impact of event management schemes on the performance of discrete-event simulation systems, a plethora of algorithms have been proposed in the last several decades [6][13][1][12][5]. In both theoretical and empirical contexts, performance of these algorithms also has been extensively investigated [2][10]. These studies were usually conducted in a general setting; usually, they assumed that applications generate event manipulation operations according to certain patterns. For instance, a model widely used in these studies is the classic hold model [16][7], in which events are enqueued and dequeued interleavingly.

In this paper, we take a different avenue. We characterize statistical properties of event operations that manifest themselves in network simulation. Based on these properties, we slightly modify the conventional calendar queue by allowing an event to carry a hidden event list that is transparent to the bucket array. Such a simple redesign leads to significant performance improvement in many circumstances. We observe two orders of execution time speedup against the conventional calendar queue in some wireline network simulations, and in wireless network simulation, the speedup scales well with the number of nodes in the topology. The new data structure is implemented and validated in the ns-2 network simulator [8], and we thus believe that it can help the networking community shorten the simulation turnaround time.

The paper is organized as follows. Section 2 gives a brief introduction to event manipulation operations in discrete event simulation and then discusses the conventional calendar queue. Section 3 presents our observations concerning the statistical property that event operations exhibit in network simulation. Based on these observations, we propose a data structure that improves the performance of conventional calendar queues; it is described in Section 4 and analyzed in Section 5. Section 6 presents the empirical results regarding the performance of our data structure. Section 7 summarizes this paper.

## 2 Background

Without exception, every discrete-event simulation system is centered on one or more data structures that manage its simulation events. Such data structures are often called *future event lists* (FELs). In the literature, other names have been used for the same concept, such as future event set, pending event set, and priority event queue. In this paper, we will use them interchangeably. In a sequential discrete-event simulator, usually a single FEL is maintained; a parallel or distributed discrete-event simulator, however, has multiple FELs, each of which is managed by a logical process in the parlance of parallel simulation.

Typical operations on a FEL include *ENQUEUE*, *DEQUEUE* and *REMOVE*. An ENQUEUE operation inserts a new event into the FEL; a DEQUEUE operation operation extracts the event that is scheduled to fire in the nearest future from the FEL; a REMOVE operation removes a specific event from the FEL. ENQUEUE and DEQUEUE operations form the basis of an event-driven simulation system. Sometimes, the REMOVE operation is also performed frequently. For instance, canceling a timer event previously scheduled is often a necessary operation in network protocol simulation.

Given the impact of event manipulation operations on simulation performance, numerous data structures have been proposed to manage events in a FEL. Among them, the calendar queue [1] has received wide recognition because of its expected $O(1)$ access time on both ENQUEUE and DEQUEUE operations under many conditions. A calendar queue, by principle, is a multi-list data structure. It divides simulation time into intervals of equal length, which are called *years*. It also maintains an array of buckets, each of which keeps a list of events in timestamp order. The multiplication of the array size $\Omega$ (i.e., the number of buckets) and the bucket width $\delta$ is the length of a year. The timestamp of an event is used to decide which bucket it should be put in. We number all the buckets from 0 to $\Omega - 1$. If an event has timestamp $t$, then the index of the bucket where it should be placed, denoted by $i$, is

$$i = \lfloor t/\delta \rfloor \bmod \Omega, \tag{1}$$

where $\lfloor \cdot \rfloor$ denotes the integer part of the inside float.

Events that are mapped into the same bucket are organized as a sorted list. When an event is enqueued, its timestamp is used to decide which bucket it should be put in and it is then inserted into the corresponding list; dequeueing an event needs to locate the event with the smallest timestamp before removing it from the list it is on.

The efficiency of calendar queue depends on the number of events in each bucket. If the number of events in the FEL is much larger or smaller than the number of buckets, its performance deteriorates significantly. It is thus important to ensure that the average length of each bucket is short. This is achieved by resizing the bucket array so that the average number of events in each bucket is not too small or too large. The complexity of a resizing operation is $O(n)$, where $n$ is the number of events in the queue.

Although calendar queues have O(1) access time under ideal operating conditions, its performance may deteriorate drastically under two unpleasant circumstances. First, because it dynamically adapts its bucket array size to the total number of events, it is possible that the bucket array size oscillates between two successive values (powers of 2 in most implementations). Second, if the distribution of event timestamps is skewed, events can be heavily clustered in a few buckets and other buckets remain empty; the data structure thus degenerates into a few sorted lists. Both cases can be detrimental to the efficiency of calendar queues.

A host of data structures have been proposed to improve the performance of conventional calendar queues. In [4], performance of calendar queues is modeled as a Markov chain and its analysis leads to some suggestions on how to select the two critical parameters in calendar queues, bucket width and bucket array size. Dynamic Lazy Calendar queues [9] measure costs associated with ENQUEUE and DEQUEUE operations and thereby adjust bucket width when necessary; they also maintain an auxiliary event list to reduce resizing overhead. Similar to Dynamic Lazy Calendar queues, SNOOPY calendar queues [14] measure performance costs on ENQUEUE and DEQUEUE operations; they, however, treat bucket width readjustment as an optimization process and thus try to derive the optimal operating parameters after a resizing operation. Ladder queues recently proposed in [15] also inherit some ideas from conventional calendar queues. They organize events hierarchically, using calendar queues as their basic elements. They do not rely on sampling heuristics to adjust bucket width; instead, they adapt to different access patterns by spawning new children calendar queues when necessary.

## 3 Motivating Observations

In previous work, performance evaluation of future event set algorithms was usually performed in a general setting. As our goal is to improve the network simulation performance, we are interested in unveiling some statistical properties of event operations that manifest themselves in network simulation. We conduct a set of experiments on two networks: one is wired and the other is wireless. The wired network is a campus network illustrated in Figure 1. It is adapted from the baseline NMS challenge topology[1]. Based on the original link delays, we add a randomized factor that is uniformly drawn between -0.01 and 0.01 to avoid synchronization among events. A server pool consisting of 4 servers is located in Net 1 (the upper right of the topology).

[1]http://www.ssfnet.org/Exchange/gallery/baseline/index.html

Each oval represents a router attached with 42 client hosts, which form 4 LANs. In total, there are 538 machines in the network. There is traffic between each client host and one of the 4 servers.

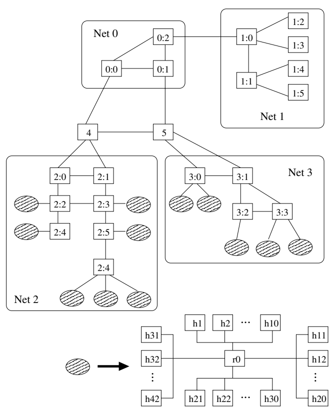


**Figure 1. Campus topology**

The wireless network we simulated is adapted from an example included in the ns-2 network simulator[2]. It has 50 nodes moving according to the random waypoint model. The routing protocol used is DSR (Dynamic Source Routing Protocol). The network interface used by each node is configured similar to the 914MHz Lucent WaveLAN DSSS radio interface. We number all the nodes from 0 to 49. There is traffic between node $i$, for $0 \leq i \leq 49$, and node $(i + 1) \bmod 50$.

We vary the types of network traffic used in both networks. The first one is CBR applications that send out traffic at a constant rate over UDP transport protocol. We put some jitter between consecutive packets. The second traffic type is large file transfers using FTP over TCP transport protocol.

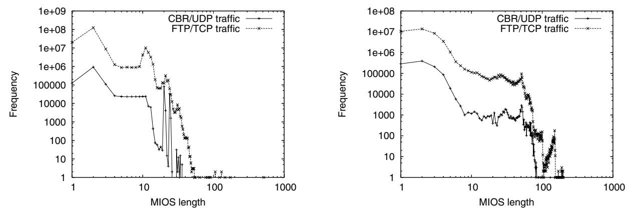
We use the ns-2 network simulator in all these experiments. In each of them, we collect the following data: as a simulation event is inserted into the FEL, we record its timestamp (i.e., the time it is scheduled to fire). After the simulation finishes, we divide the final series of event timestamps into the longest sequences with non-decreasing timestamps, which are called *maximal in-order sequences* and abbreviated as *MIOSs*. For example, suppose that we have the following data:

$$0.5 \quad 0.4 \quad 0.7 \quad 0.9 \quad 2.1 \quad 1.4 \quad 3.5 \quad 3.5 \quad 3.6 \quad 3.1,$$

each of which indicates the timestamp of an inserted event. They thus contain 4 MIOSs:

$$[0.5], [0.4 \quad 0.7 \quad 0.9 \quad 2.1], [1.4 \quad 3.5 \quad 3.5 \quad 3.6], [3.1].$$

[2]The input file is ns-2.29/tcl/ex/wireless.tcl



(1) Wireline network (2) Wireless network

**Figure 2. Frequency histogram on MIOS length.**

Figure 2 presents the frequency histogram on the MIOS lengths in all the experiments. Note that the figure uses logarithmic scale. In all four cases a large number of MIOSs involve more than one event, and a few MIOSs even have more than 100 events in them! The following table gives the percentage of MIOSs that have at least 5 and 10 events in all the scenarios:

| | Wireline | | Wireless | |
|---|---|---|---|---|
| | CBR | FTP | CBR | FTP |
| $\geq 5$ events | 20.1% | 16.5% | 7.6% | 11.3% |
| $\geq 10$ events | 14.1% | 14.1% | 4.8% | 6.6% |

The long sequences of in-order events observed in the network simulation can originate from several places. First, shared media have been widely used in both wireline and wireless networks. On such media, a packet transmitted by one entity can be seen by all other parties. Typical implementations usually use a simulation event to signal the arrival of this packet at each receiver. Hence all the simulation events triggered by a single packet transmitted bear the same timestamp (if we do not put any jitter on delays. In many circumstances jitter does not affect simulation results significantly.) Second, bursty traffic such as TCP traffic leads to in-order events. In packet-level traffic simulation, packet arrivals are usually represented as simulation events. If traffic is more bursty, intervals between packet arrivals are shorter and it is thus more likely that events representing them are inserted into the FEL consecutively. In addition, bursty traffic not only results from bursty traffic sources like TCP applications, but also comes from congested network components such as queues in NICs (Network Interface Cards). Finally, many scheduling algorithms in network protocols use FIFO (First In First Out) mechanism. Therefore, after old events representing input traffic are processed, new events generated to represent output traffic are still in timestamp order.

As we have observed above, a significant number of events in network simulation appear in timestamp order when they are inserted into the FEL. A natural question, then, is: can we exploit such an observation to improve the performance of pending event set algorithms? In the next section, we describe a *sluggish calendar queue*, a modified version of the conventional calendar queue, that exploits in-order event sequences to accelerate event manipulation operations.

## 4 Sluggish Calendar Queues

A sluggish calendar queue consists of two components: a *layman event list* and a bucket array. The layman event list is a doubly-linked list, which stores events in timestamp order. We call an event on the layman list a *layman* event. The bucket array resembles the bucket array in a conventional calendar queue but events are organized in a different manner. In each bucket, all the events that have been mapped into it are organized into a doubly-linked list in timestamp order. We call such a list a *trunk event list* and an event on a trunk event list a *trunk event*. In contrast to the conventional calendar queue, a trunk event in a sluggish calendar queue keeps a pointer to another doubly-linked list, which is called a *branch event list*. An event on a branch event list is called a *branch event*. Events on a branch event list are also organized in timestamp order, but their timestamps may not necessarily be mapped into the same bucket as the corresponding trunk event. The timestamp of a trunk event must be no greater than that of any event on its branch event list.

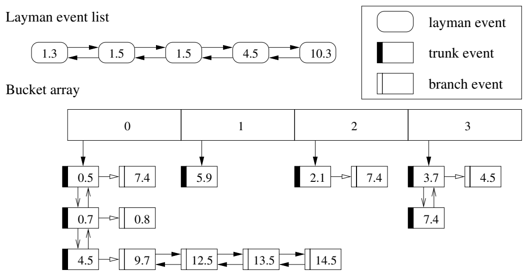


**Figure 3. A sluggish calendar queue**

Figure 3 illustrates a sluggish calendar queue with 4 buckets and the width of each bucket is 1 time unit. In this queue, there are 20 events in total but there are only 7 trunk events. Hence the number of trunk events can be much smaller than the total number of events in the queue. Later we will explain that this is the key idea of this data structure. In addition, the trunk event with timestamp 0.5 in bucket 0 has only one event on its branch event list, but that event, with timestamp 7.4, should be mapped into bucket 3 if it were a trunk event.

We now describe event manipulation operations on a sluggish calendar queue. The data structure is extended from the conventional calendar queue. Thus, event manipulation operations on these two data structure bear a lot of similarities. In the following discussion, we will highlight the places where the sluggish calendar queue differs from the conventional calendar queue. We ask readers to refer to [1] for details of conventional calendar queues.

*Stability* is an important property of pending event set algorithms. A stable event manipulation algorithm preserves the enqueueing order of any two events carrying the same timestamp when they are dequeued. Stable event manipulation algorithms are often desirable because they facilitate debugging of simulation code. We thus strictly impose the stability requirement upon the sluggish calendar queue.

### 4.1 ENQUEUE Operation

When we insert a new event, denoted by $e_{new}$, into a sluggish calendar queue, we first check whether it can be added onto the layman event list. We use parameter $\alpha$ to constrain the number of events we want to check against the new event. Starting from the tail of the layman event list, we iteratively compare the timestamps of the last $\alpha$ events against that of the new event; if we find one that has a *smaller* timestamp than the latter or an *exactly the same* timestamp as the latter, we stop the iteration and insert the new event after that event. If we fail to find the proper position among the last $\alpha$ events, we check the first $\alpha$ events. Similarly, starting from the head of the layman event list, we iteratively compare the timestamps of the first $\alpha$ events against that of the new event; if we find one that has a *larger* timestamp than the latter, we stop the iteration and insert the new event before that event. We are careful here to ensure that events bearing the same timestamp appear on the layman event list in a FIFO manner. In our implementation, we set the default value of $\alpha$ to be 3.

In the above approach, if there are less than $2\alpha$ events on the layman event list, some events may be compared against the new one twice. To avoid that, we keep the total number of events on the layman event list, denoted as $L_{lel}$. Before all the comparisons above, we check whether $L_{lel}$ is larger than $2\alpha$. If it is, we perform the aforementioned comparisons; otherwise, we simply check all the events on the list to find the proper location. If the new event is inserted onto the layman event list, we increase $L_{lel}$ by one.

It is possible that the new event can not be placed among the first $\alpha$ or the last $\alpha$ events on the layman event list. If this is true, we migrate the layman event list into the bucket array. More specifically, the head event of the layman event list is converted to a trunk event and all other events to its branch events. Based on the timestamp of the new trunk event, we use Equation (1) to decide which bucket it should be put in and then insert it onto the corresponding trunk event list in timestamp order. To ensure the FIFO property described earlier, all the events that bear the same timestamp must appear before the new trunk event. After migrating the whole layman event list into the bucket array, we add the new event to the layman event list. At this time, the layman event list has a single event on it.

We continue working on the example shown in Figure 3.

Suppose that we need to enqueue two events, whose timestamps are 11.7 and 1.7 respectively. For the first one, we simply put it at the tail of the layman event list. For the second, however, we need to migrate the layman event list into the bucket array. As the head event has timestamp 1.3, the new trunk event should be inserted into bucket 1. Figure 4 illustrates the new layman event list and the structure of bucket 1 after all the operations. The other buckets remain intact and are thus ignored in the graph.

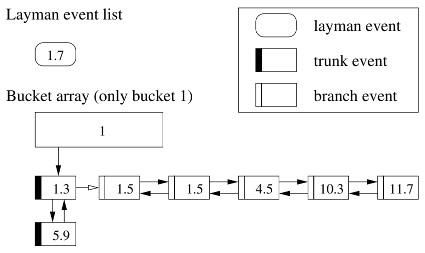


**Figure 4. The sluggish calendar queue after enqueueing operations**

After a new trunk event is inserted into the bucket array, it is checked whether the number of buckets in the array needs to be readjusted. We leave more detailed discussion on resizing operations in Section 4.4.

## 4.2 DEQUEUE Operation

The DEQUEUE operation extracts the event that bears the smallest timestamp from the sluggish calendar queue. Because the timestamp of a trunk event must be no greater than that of any event on its branch event list, we can ignore all branch events when deciding which event to be dequeued next. It, however, is possible that the event with smallest timestamp is still on the layman event list. As events on the layman event list have already been ordered in timestamps, the event to be dequeued should be either the head event of the layman event list or the trunk event with the smallest timestamp in the bucket array.

When we dequeue an event from a sluggish calendar queue, we first use the same method as in the conventional calendar queue to locate the trunk event with the smallest timestamp, and then compare it against the timestamp of the head event on the layman event list. If the former is *greater than* the latter, we simply remove the head event from the layman event list. Here, we notice that an event on the layman event list must be enqueued *after* any one that has already been migrated into the bucket array. Hence, if the two events have exactly the same timestamp, we should dequeue the trunk event first based on the FIFO policy. If the trunk event has a smaller timestamp, we should also dequeue the trunk event first.

When we dequeue a trunk event, we first remove it from the corresponding trunk event list. If the trunk event has a branch event list, we convert the head event of the branch event list to a trunk event and the remaining part of the list to the branch event list of the new trunk event, and then insert the new trunk event into the proper bucket in timestamp order. Unfortunately, if the trunk event list has an event that bears exactly the same timestamp as the new trunk event, we can not use the FIFO policy to decide their orders any more. An example can help us understand this situation. Suppose four events are enqueued in order and their timestamps are 0.8, 1.3, 0.7, and 1.3 respectively. They form two MIOSs, each having two events. We also assume that both of them have been migrated into the bucket array, the bucket width of which is 1 simulation time unit. After the event with timestamp 0.7 is dequeued, the last event with timestamp 1.3 is inserted into the trunk event list in bucket 1; later after the event with timestamp 0.8 is dequeued, we need to insert its branch event with timestamp 1.3 into bucket 1. If we apply the FIFO policy, this event should be put *behind* the one that is already there. This obviously violates the order in which they were enqueued. We thus can not use the FIFO policy to decide the order in which events with the same timestamp should be positioned in the trunk event list.

To solve this problem, we assign a unique *MIOS identifier* to each trunk event when it is migrated from the layman event list. When we dequeue a trunk event, we pass its identifier on to the new trunk event converted from its branch event list. Apparently, no two trunk events have the same MIOS identifier in the system. In the previous setting when we insert a new trunk event onto a trunk event list, if it has the same timestamp as another one already on the list, we use their MIOS identifiers for tie-breaking. Events with smaller MIOS identifiers are placed before those with larger values on the trunk event list.

MIOS identifiers may be exhausted if the simulation runs for a long time. When this occurs, we can reassign MIOS identifiers to the trunk events in the bucket array. A simple reassignment scheme is to sort the MIOS identifiers of all the trunk events and then reassign the MIOS identifier of a trunk event as the order in the sorted sequence. Hence, the original order of MIOS identifiers is still maintained. The complexity of this approach is $O(m \cdot \log m)$, where $m$ is the number of trunk events at the time of reassignment.

If MIOS identifiers are exhausted quickly and reassignment is an expensive process, we can concatenate all trunk events into a doubly-linked list based on their orders of MIOS identifiers. We call it the *universal trunk event list*. When migrating the layman event list into the bucket array, we put the new trunk event at the tail of the universal trunk event list. When a trunk event is dequeued, if one of its branch events is converted to a trunk event, we substitute the new trunk event for the old one on the universal trunk event list and otherwise, we remove the old trunk event from the list. The universal trunk event list can be used to speed up reassignment process. Because events appear on the list in the same order as they are enqueued, the reassignment process becomes simple: the MIOS identifier of each trunk event is reset to be its position on this list. In this approach, the reassignment process takes only $O(m)$ time, where $m$ is the number of trunk events in the bucket array. The performance improvement comes at a price of extra memory required to maintain the universal trunk event list.

Back to the example we discussed earlier, suppose a DEQUEUE operation is performed on the sluggish calendar queue shown in Figure 3. The event with timestamp 0.5 has the smallest timestamp among all the events. After it is dequeued, its single branch event with timestamp 7.4 becomes a trunk event, which is mapped into bucket 3. As that bucket already has a trunk event with timestamp 7.4, we need to use their MIOS identifiers to decide their relative order. The new sluggish calendar queue is illustrated in Figure 5.

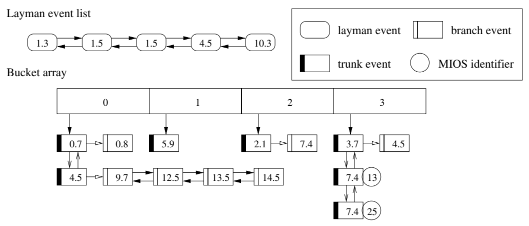


**Figure 5. The sluggish calendar queue after dequeueing an event**

If a trunk event is dequeued from the bucket array, we check whether its size needs to be readjusted. The resizing operation is discussed in detail in Section 4.4.

## 4.3 REMOVE Operation

The REMOVE operation is very important in network simulation. For example, when a TCP packet is transmitted, a retransmission timer is scheduled. If the packet is successfully received and thus acknowledged by the receiver, the retransmission timer scheduled earlier needs to be canceled. A REMOVE operation is thus performed to erase the given event from the FEL. When a REMOVE operation is called, a pointer to the event is also specified.

In a sluggish calendar queue, removing an event is simple because all events are organized in doubly-linked lists. If the event to be deleted is a layman event, we simply remove it from the layman event list. Similarly, removing a branch event just needs to erase it from the corresponding branch event list. Removing a trunk event is exactly the same as dequeueing a trunk event: erase it from the trunk event list, convert the head event on its branch event list to a trunk event and then put the new trunk event into the corresponding bucket. In addition, after removing a trunk event, we need to check whether the bucket array should be resized.

## 4.4 RESIZE Operation

Similar to the conventional calendar queue, the efficiency of a sluggish calendar queue largely depends on how trunk events are distributed in the buckets. If there are too many trunk events in a bucket, it may take a significant amount of time to insert a new trunk event onto its trunk event list; on the other hand, if there are too few trunk events in the bucket array, many buckets have empty trunk event lists. Hence, we need to readjust the array size under both circumstances.

Let $N_{tr}$ denote the number of trunk events in the system and the $\Omega$ be the array size. The condition for triggering an operation that increases the array size is defined as

$$N_{tr} > \lambda_h \cdot \Omega. \quad (2)$$

Similarly, the condition for triggering an operation that reduces the array size is defined as

$$N_{tr} < \lambda_l \cdot \Omega. \quad (3)$$

In the implementation of the ns-2 network simulator, $\lambda_h$ is 2 and $\lambda_l$ is 1/2 (in a conventional calendar queue, all events can be deemed as trunk events). However, a sluggish calendar queue can tolerate relatively few trunk events. Even though some buckets may have empty trunk event lists, it is possible that as trunk events are dequeued and thus their branch events are converted to trunk events, new trunk events may be mapped onto these buckets. Therefore, the performance of sluggish calendar queues is less sensitive to relatively few trunk events. We thus reduce the possibility of decreasing bucket array size by using smaller $\lambda_l$, whose default value is 1/16 in our implementation.

Performance of a calendar queue is a function of both the total number of buckets $\Omega$ and the bucket width $\delta$ [4]. As simulation advances, it is possible that an improper bucket width may cause trunk events to be clustered in only a few buckets. This may also happen to a sluggish calendar queue. To avoid this, we force a sluggish calendar queue to readjust its bucket width if the number of trunk events in a bucket exceeds a threshold $\theta$. In our implementation, we set $\theta$ to be 100 by default. We use a relatively large threshold to avoid readjusting $\delta$ frequently at the initial stage of a simulation when usually many events are enqueued consecutively. Once the total number of the events in the system becomes stationary, we can decrease it to a smaller value.

In a RESIZE operation, we readjust $\delta$ in the same way as the implementation of calendar queues in the ns-2 network simulator. It is briefly introduced as follows. First, we check the most populated bucket, whose index is denoted by $\psi$, and let $T_{min}$ and $T_{max}$ be the minimum and maximum timestamps among all the trunk events in that bucket respectively. Because all trunk events in a bucket are organized into a cyclic linked list in the implementation, only constant time is needed to determine $T_{min}$ and $T_{max}$. We use $\widetilde{n}$ to denote the number of trunk events with different timestamps in bucket $\psi$. Then the new bucket width is determined by the following formula:

$$\delta = \frac{4(T_{max} - T_{min})}{\min(\Omega, \widetilde{n})}. \tag{4}$$

After the bucket array is resized or the bucket width is readjusted, we enqueue all the events in the old bucket array into the new one.

## 5 Algorithm Analysis

In this section, we analyze the event manipulation operations of a sluggish calendar queue from both correctness and performance perspectives.

### 5.1 Correctness Analysis

Correctness of an event-driven simulation system is relevant to whether *causality errors* occur. In the context of pending event set algorithms, we define causality errors as situations in which events are dequeued from the FEL out of timestamp order. Occurrence of causality errors in this setting indicates incorrectness of event manipulation operations and is thus not allowable. Based on the algorithm description in Section 4, we can easily prove the following theorem (proof is omitted here):

**Theorem 1** *Sluggish calendar queues do not produce causality errors.*

Similarly, we have the following theorem (proof omitted):

**Theorem 2** *Sluggish calendar queues are stable.*

### 5.2 Performance Analysis

Complexity analysis of sluggish calendar queues is similar to that of conventional calendar queues. For a majority of ENQUEUE and DEQUEUE operations, it is unnecessary to resize the bucket array size and they can thus be finished in $O(1)$ time. If a RESIZE operation is performed when an event is enqueued or dequeued, $O(m)$ time is needed, where $m$ is the total number of *trunk events* in the system. Therefore, the expected amortized cost associated with an ENQUEUE (or DEQUEUE) operation is $O(1)$ under normal operating conditions, but in the worst case where almost every ENQUEUE (or DEQUEUE) operation needs to resize the bucket array size, the amortized cost becomes $O(m)$.

Here we notice the performance difference between sluggish calendar queues and conventional calendar queues. For the latter, although its expected amortized cost associated with an ENQUEUE (or DEQUEUE) operation is also O(1), its worst case amortized cost is $O(n)$, where $n$ is total number of events in the system [10]. Therefore, if a simulation system like many network simulations has a lot of events enqueued in non-decreasing timestamp order (i.e., $n >> m$), sluggish calendar queues suffer less performance degradation than conventional calendar queues under unpleasant circumstances.

As the number of trunk events in the system affects the performance of sluggish calendar queues, we analytically derive its relationship with the total number of events. For tractability purpose, we assume that all MIOSs have the same size $w$. We also assume that no layman event is dequeued; in other words, when an event is dequeued, it must be a trunk event. Furthermore, the probability of dequeueing a trunk event is uniformly distributed over all the trunk events in the system. The pattern the sluggish calendar queue is accessed is modeled by the classic hold model, in which ENQUEUE and DEQUEUE operations are invoked interleavingly [16][7]. Hence, the total number of events in the system after a pair of ENQUEUE and DEQUEUE operations remain constant. The system is initialized to have $n$ events.

We then do steady state analysis on the system as described. Let $a_i^{(0)}$, for $1 \leq i \leq w$, denote the number of trunk events with $i-1$ branch events at the steady state. We use $A^{(0)}$ to represent $\Sigma_i a_i^{(0)}$. We consider the distribution of trunk events after $k$ ($k \geq 1$) pairs of DEQUEUE and ENQUEUE operations . Let $b_i^{(k)}$ denote the expected number of trunk events with $i-1$ branch events after the $k$-th DEQUEUE operation and $a_i^{(k)}$ the expected number of trunk events with $i-1$ branch events after the $k$-th ENQUEUE operation. Similarly, let $A^{(k)}$ represent $\Sigma_i a_i^{(k)}$. Consider the $k$-th DEQUEUE operation. Because the trunk event to be dequeued is uniformly distributed, the probability of choosing a trunk event that has $w-1$ branch events is $a_w^{(k)}/A^{(k)}$. Hence, we have

$$\begin{aligned} b_w^{(k)} &= (1 - a_w^{(k)}/A^{(k)}) \times a_w^{(k)} + a_w^{(k)}/A^{(k)} \times (a_w^{(k)} - 1) \\ &= (1 - 1/A^{(k)}) \times a_w^{(k)}. \end{aligned} \tag{5}$$

Similarly, we have the following:

$$\begin{aligned} b_{w-1}^{(k)} &= (1 - 1/A^{(k)}) \times a_{w-1}^{(k)} + a_w^{(k)}/A^{(k)} && (6) \\ b_{w-2}^{(k)} &= (1 - 1/A^{(k)}) \times a_{w-2}^{(k)} + a_{w-1}^{(k)}/A^{(k)} && (7) \\ &\dots \\ b_1^{(k)} &= (1 - 1/A^{(k)}) \times a_1^{(k)} + a_2^{(k)}/A^{(k)} && (8) \end{aligned}$$

Comparing Equations (5) and (6), we notice that the latter has one more term. That results from the case in which a trunk event with $w-1$ branch events is dequeued and thus a new trunk event with $w-2$ branch events is generated. This observation also applies to the other equations.

After the $k$-th ENQUEUE operation, it is possible that the layman event list is converted to a trunk event with $w-1$ branch events. For each ENQUEUE operation, this probability is $1/w$. Hence, we have

$$\begin{aligned} a_w^{(k)} &= b_w^{(k)} \times (1-1/w) + (b_w^{(k)}+1) \times 1/w \\ &= b_w^{(k)} + 1/w. \end{aligned} \tag{9}$$

For trunk events with any other number of branch events, their numbers remain the same after an ENQUEUE operation. Therefore,

$$a_t^{(k)} = b_t^{(k)}, \qquad \forall t : 1 \le t \le w-1. \tag{10}$$

If we assume that the system is steady, its state must return to the original state after some finite number of iterations, which is denoted as $K$. We then build a system of equations as follows:

$$\left\{ \begin{array}{lcl} a_w^{(K)} &=& a_w^{(0)} \\ a_{w-1}^{(K)} &=& a_{w-1}^{(0)} \\ & \cdots & \\ a_1^{(K)} &=& a_1^{(0)}. \end{array} \right. \tag{11}$$

Unfortunately, it is very difficult to solve the above system of equations analytically. If we are limited to those systems where the number of trunk events is much larger than $w$, then $A^{(k)}$ is relatively constant. We assume that $A^{(k)}$ is $m$ for all $k$ ($k \ge 0$). We then derive the following from Equation (11):

$$a_t^{(0)} = m/w, \qquad \forall t : 1 \le t \le w. \tag{12}$$

Equation (12) suggests that in the steady state, there is the same number of trunk events with different branch event list lengths. Recall that the total number of events in the system is $n$. On average there are $w/2$ events on the layman event list. We thus have

$$n = w/2 + (1+2+...+w) \cdot m = w/2 + \frac{w(w+1)}{2} \cdot m \tag{13}$$

Therefore,

$$m = \frac{2n-w}{w(w+1)}. \tag{14}$$

Following the above result, we can establish the following theorem:

**Theorem 3** *Given the assumptions on access patterns as described, sluggish calendar queues reduce the worst case amortized cost on ENQUEUE and DEQUEUE operations by a factor of $\Theta(w^2)$ relative to conventional calendar queues, where $w$ is the number of events in a MIOS.*

## 6 Experiments

We have implemented the sluggish calendar queue in the ns-2 network simulator. To ensure a fair comparison between them in later experiments, we first validate our implementation against the conventional calendar queue. Since both pending event set algorithms are stable, access patterns on the future event list should be exactly the same if the simulation scenario is the same. This is consistently observed in our validation tests.

We now investigate how sluggish calendar queues perform empirically with a new set of experiments, which extend the topologies used in Section 3. We still use the two types of traffic, FTP traffic over TCP and CBR traffic over UDP. In the wireline network simulation, we vary the number of campus networks between 1, 2, 3, and 4. If there is only one campus network, there is traffic between every client and one of the four servers in it; otherwise, all the campus networks form a ring topology and traffic is specified between each client in a campus network and a server in the next campus network clockwise. In the wireless network simulation, we vary the number of mobile nodes between 50, 100, and 150. In the scenario with $N$ nodes, we number the nodes from 0 to $N-1$. The traffic pattern is defined between every node $i$ and node $(i+1)$ mod $N$.

### 6.1 Bucket Array Snapshots

We take the snapshots of the bucket arrays in some experiments. Figure 6 gives the results from simulating the wireline networks with 2 and 4 campuses. The left side corresponds to the conventional calendar queue and the right side to the sluggish calendar queue. One observation is that the bucket array size using the sluggish calendar queue is only a half or one fourth of that using the conventional calendar queue. This is unsurprising, since for the sluggish calendar queue branch events and layman events are not included in the figure. We are more interested in how (trunk) events are distributed in the buckets because this largely determines the average cost associated with an ENQUEUE operation. It is clear that the number of events in the buckets under the conventional calendar queue exhibits higher variation than the number of trunk events in the buckets under the sluggish calendar queue. This is especially prominent for TCP traffic simulation. When 2 campuses are simulated, bucket 2324 contains 367 events, much more than any other bucket. More interestingly, when 4 campuses are simulated, the calendar queue essentially degenerates into only a few linearly sorted lists: bucket 152 has 10548 events, bucket 153 has 307 events, bucket 154 has 37 events, bucket 242 has 896 events, and almost all other buckets do not have any events in them. If the sluggish calendar queue is used, however, we do not observe this: in all the experiments, the longest trunk event list has only 12 events on it.

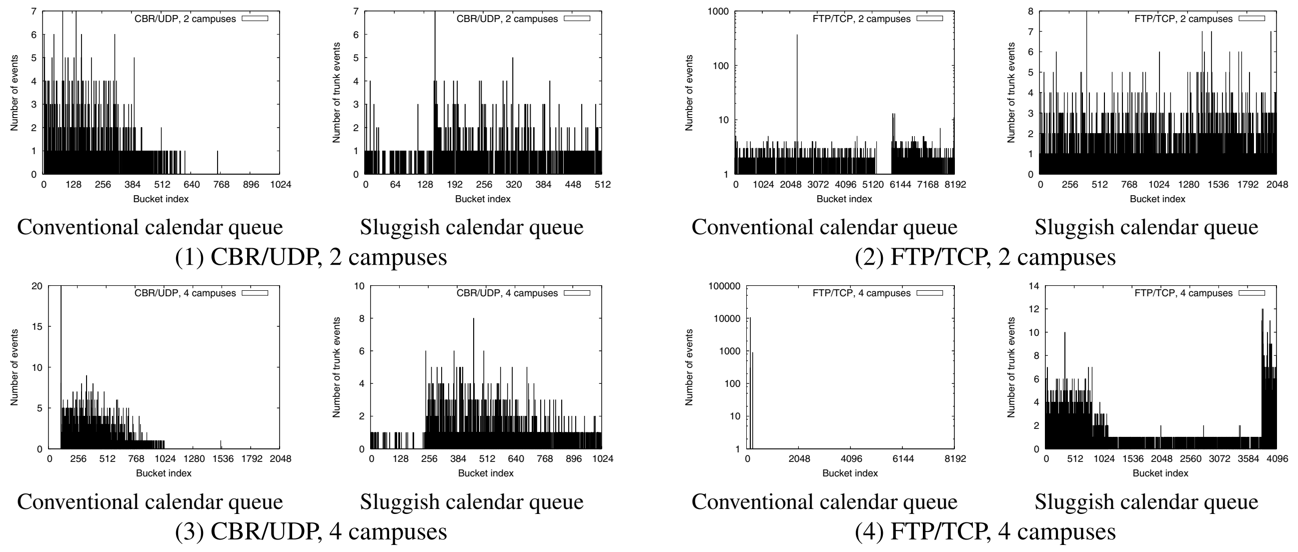


**Figure 6. Number of (trunk) events in each bucket in the wireline network simulation.**

Figure 7 presents the bucket array snapshots from simulating the wireless networks with 100 and 150 nodes. Similarly, the left side corresponds to the conventional calendar queue and the right side to the sluggish calendar queue. Observations made from the wireline network simulation repeat themselves in the wireless network simulation. The number of buckets under the conventional calendar queue is much larger than that under the sluggish calendar queue; the ratio between these two varies between 256 and 512. The number of events in the buckets under the conventional calendar queue still has higher variation than the number of trunk events in the buckets under the sluggish calendar queue. Compared with the wireline network simulation, however, events are more evenly spread among all the buckets except that for the scenario having TCP traffic and 100 nodes, events seem clustered in the first half part of the bucket array. Under the sluggish calendar queue, the peak number of trunk events in a bucket is still very small, which is 13. Another interesting observation is that when the number of mobile nodes increases, trunk events tend to spread in the buckets more evenly.

## 6.2 Speedup

We simulate each topology for 10 times on a standalone (i.e., without network connection) desktop with 3GHz CPU and 1GB memory. The simulation length for each topology is set long enough to reduce the impact of other factors such as background OS programs. Figure 8 depicts the *speedup* of the sluggish calendar queue in both wireline and wireless network simulations. The speedup is defined as the average execution time needed if the conventional calendar queue is used divided by that if the sluggish calendar queue is used.

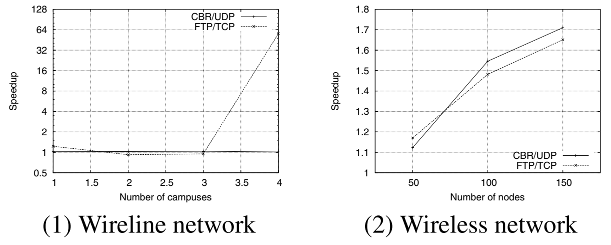
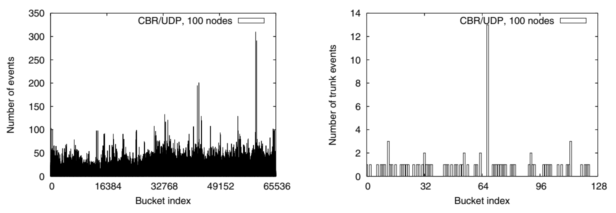


**Figure 8. Execution speedup.**

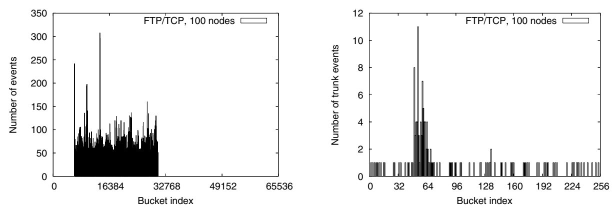
From Figure 8, we observe that for the wireless CBR traffic simulation, the sluggish calendar queue performs slightly better than the conventional calendar queue. In all cases, the average speedup is 1.024. On the other hand, for the wireline FTP traffic simulation, the speedup varies with the number of campuses in the network. If there is only one campus, the speedup is 1.226. If there are two or three campuses, the sluggish calendar queue performs slightly worse than the conventional calendar queue, with average speedup 0.934; we conjecture that this results from the extra computation cost imposed by the sluggish calendar queue, such as event comparison on the layman event list. More interesting is the case when there are 4 campuses: the speedup is as high as 56! From Figure 6, we know that in this scenario the bucket array in the conventional calendar queue degenerates into a few sorted lists, which obviously slows down the simulation significantly.

The speedups for wireless network simulations are more consistent, as shown in Figure 8. When we increase the number of nodes from 50 to 150 in the topology, the speedup grows monotonically from about 1.1 to 1.7, irrespective of the traffic type in the network. But we do not



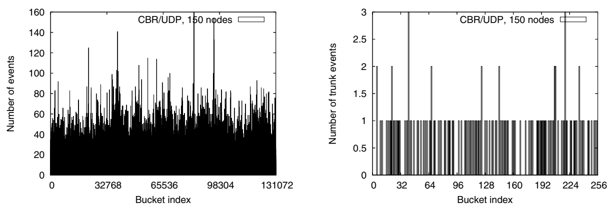
Conventional calendar queue Sluggish calendar queue

(1) CBR/UDP, 100 nodes



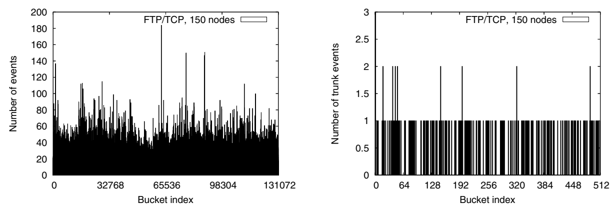
Conventional calendar queue Sluggish calendar queue

(2) FTP/TCP, 100 nodes



Conventional calendar queue Sluggish calendar queue

(3) CBR/UDP, 150 nodes



Conventional calendar queue Sluggish calendar queue

(4) FTP/TCP, 150 nodes

**Figure 7. Number of (trunk) events in each bucket in the wireless network simulation.**

observe the dramatic performance improvement as seen in the 4-campus wireline network simulation.

## 7 Conclusions

In this paper, we have shown that in network simulation events are often inserted into the FEL in their timestamp order. The sluggish calendar queue rests on this observation and extends the conventional calendar queue by allowing events to carry a branch event list. Experiments show that the new data structure performs much better than the conventional calendar queue in many cases and comparably well in others. For future research direction, we plan to test sluggish calendar queues in other application domains, such as discrete-event social activity simulations.

## References

[1] R. Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10), 1988.

[2] K. Chung, J. Sang, and V. Rego. A performance comparison of event calendar algorithms: An empirical approach. *Software-Practice and Experience*, 23(10), October 1993.

[3] J. C. Comfort. The simulation of a microprocessor based event set processor. In *Proceedings of the 14th Annual Symposium on Simulation*, Tampa, Florida, USA, 1981.

[4] K. B. Erickson and R. E. Ladner. Optimizing static calendar queues. *ACM Transactions on Modeling and Computer Simulation*, 10(3), July 2000.

[5] R. S. M. Goh and I. L-J Thng. Mlist: An efficient pending event set structure for discrete event simulation. *International Journal of Simulation*, 4(5-6), December 2003.

[6] G. H. Gonnet. Heaps applied to event driven mechanisms. *Communications of the ACM*, 19(7), 1976.

[7] D. W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4), April 1986.

[8] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[9] S. Oh and J. Ahn. Dynamic lazy calendar queue: An event list for network simulation. In *Proceedings of the 32nd Annual Simulation Symposium*, 1999.

[10] R. Rönngren and R. Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation*, 7(2), April 1997.

[11] G. R. Riley and M. H. Ammar. Simulating large networks - how big is big enough? In *Proceedings of First International Conference on Grand Challenges for Modeling and Simulation*, January 2002.

[12] R. Ronngren, J. Riboe, and R. Ayani. Lazy queue: An efficient implementation of the pending-event set. In *Proceedings of the 24th Annual Simulation Symposium*, 1991.

[13] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3), July 1985.

[14] K. L. Tan and L.-J. Thng. SNOOPY calendar queue. In *Proceedings of the 2000 Winter Simulation Conference*, 2000.

[15] W. T. Tang, R. S. M. Goh, and I. L.-J. Thng. Ladder queue: An O(1) priority queue structure for large-scale discrete event simulation. *ACM Transactions on Modeling and Computer Simulation*, 15(3), July 2005.

[16] J. G. Vaucher and P. Duval. A comparison of simulation event lists. *Communications of the ACM*, 18(4), June 1975.

[17] G. Yan. *Improving Large-Scale Network Traffi c Simulation with Multi-Resolution Models*. PhD thesis, Department of Computer Science, Dartmouth College, 2005.