Ladder Queue: An $O(1)$ Priority Queue Structure for Large-Scale Discrete Event Simulation

WAI TENG TANG, RICK SIOW MONG GOH, and IAN LI-JIN THNG
National University of Singapore

This article describes a new priority queue implementation for managing the pending event set in discrete event simulation. Extensive empirical results demonstrate that it consistently outperforms other current popular candidates. This new implementation, called Ladder Queue, is also theoretically justified to have $O(1)$ amortized access time complexity, as long as the mean *jump* parameter of the priority increment distribution is finite and greater than zero, regardless of its variance. Many practical priority increment distributions satisfy this condition including unbounded variance distributions like the Pareto distribution. This renders the LadderQ the ideal discrete event queue structure for stable $O(1)$ performance even under practical queue distributions with infinite variance. Numerical simulations ranging from 100 to 10 million events affirm the $O(1)$ property of LadderQ and that it is a superior structure for large-scale discrete event simulation.

Categories and Subject Descriptors: E.1 [**Data**]: Data structures; F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*discrete event*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Pending event set implementations, priority queue, calendar queue

1. INTRODUCTION

The Sorted-discipline Calendar Queue (SCQ), as proposed by Brown [1988], is an important implementation of the pending event set (PES) in discrete event simulators such as GTW [Das et al. 1994], CSIM18 [Schwetman 1996], and Network Simulator v2 [Fall and Varadhan 2002]. By far, SCQ is more popular than the unsorted bucket discipline calendar queue, or UCQ. This is because the number of operations required for a basic enqueue and dequeue operation is $O(n_{sublist})$ and $O(1)$, respectively, in a SCQ bucket where $n_{sublist}$ represents

Authors' address: Department of Electrical and Computer Engineering, National University of Singapore, 3 Engineering Drive 3, Blk E4A #05-06, Singapore 117576; email: {waitengtang,smgoh, eletlj}@nus.edu.sg.

the number of events in an SCQ bucket. For the case of the UCQ, the number of operations required is higher, that is, $O(1)$ for enqueue and guaranteed $n_{sublist}$ operations for dequeue.

Rönngren and Ayani [1997] provided empirical evidence that the SCQ achieves $O(1)$ performance in benchmark scenarios where the number of events N in the queue and the mean of the *jump* random variable associated with the priority increment distribution, denoted as μ, remain constant. However, the SCQ sometimes falters to $O(n)$ when μ varies, for example, the Camel and Change distributions, even though N is constant. Furthermore, when N varies, the performance of the SCQ becomes erratic with many peaks which occur when the queue size fluctuates by a factor of 2, revealing that the resize operations can be costly. In addition, when N varies, the SCQ does not always achieve $O(1)$ performance even though μ remains constant, for example, Triangular distribution. These observations translate to the following:

(a) The SCQ's size-based resize trigger is an ineffective mechanism for handling skewed distributions where μ varies. It results in many events being enqueued into a few buckets with long sublists and many empty buckets (i.e., skewed distribution phenomena). Long sublists make enqueue operations expensive since each enqueue entails a sequential search, whereas excessive traversal of empty buckets can increase the process of dequeue operations. This problem of the SCQ has been attributed to the size-based resize triggers which are too rigid to react according to the events distribution encountered since a resize trigger occurs only if the queue size halves or doubles [Brown 1988].

(b) Size-based resize trigger is not suitable for handling simulation scenarios where N varies by a factor of two frequently. This form of trigger is considered inflexible because even though the SCQ can be performing well with its existing operating parameters, but due to this trigger, it still has to resize when N varies by a factor of two.

(c) Sampling heuristic is inadequate to obtain good operating parameters, namely, the number of buckets and the bucketwidth, when skewed distributions are encountered. During each resize operation, the SCQ samples, at most, the first twenty-five events. This is clearly too simplistic because for skewed distributions in which many events fall into some few buckets, the inter-event time-gap of the first twenty-five events, which can span several buckets, and those in the few populated buckets may vary a lot. To simply increase the events sampled is not a prudent approach unless the distribution is uniform. For most distributions, particularly skewed distributions such as the Camel distribution, if we simply sample more events and then take the mean or median, it is unlikely that it will be accurate since events are spread unevenly. Even if the most populated bucket is sampled, the SCQ will also perform poorly because events in that bucket will have a small average time-gap whereas other events have widely diverse time-gaps. If the bucketwidth is updated to this small time-gap, there will likely be numerous empty buckets. And skipping these empty buckets will lead to inferior performance. Furthermore, sampling more events inevitably leads to higher overheads for each resize operation, affecting the SCQ performance on a whole.

(d) Events are immediately sorted in all the bucket sublists as they are enqueued. This enqueue process is inefficient and costly if the event distribution is heavily skewed so that bucket sublists are very long resulting in costly linear searches during enqueue operations. Furthermore, if resize operations are triggered, the effort of sorting these events previously are all wasted since these events will have to be re-enqueued and re-sorted again. Such resize operations are very costly and time consuming in unstable large-scale scenarios where N varies, resulting in resize triggers being activated many times.

In this article, we introduce a novel multilist-based priority queue structure which overcomes the above problems to achieve $O(1)$ performance in which N and/or μ vary. These include skewed distributions as well as heavy-tailed distributions such as the Pareto with finite mean but unbounded variance. We coin this structure the Ladder Queue (LadderQ). The reason for using a multilist as the basic structure for LadderQ rather than a tree-based structure is that tree-based structures are known to be bounded by $O(\log(n))$ whereas multilist-based structure such as the SCQ have an expected $O(1)$ performance except for scenarios in which the μ and/or N vary with time. However, unlike the SCQ, the LadderQ is insensitive to its bucketwidth parameter δ, as well as insensitive to N. In fact, we justify theoretically that the LadderQ is $O(1)$ in both its enqueue and dequeue operations given that the mean of μ is finite and greater than zero, regardless of its variance. Unlike the theoretical UCQ studied in Erickson et al. [2000], we do not impose restrictive or sterile conditions, like δ must vary according to $O(1/N)$, in order to show that the LadderQ is $O(1)$ theoretically. In addition, we present empirical evidence obtained from rigorous experimental studies to illustrate that the LadderQ exhibits $O(1)$ *amortized* [Tarjan 1985] (or average) complexity for widely-varying priority increment distributions, as well as for queue size ranging from 100 to 10 million. These verifications establish the necessary evidence that the LadderQ structure is superior compared to all current PES implementations.

The rest of this article is organized as follows: Section 2 describes in detail the LadderQ algorithm; Section 3 provides the theoretical proof of LadderQ's $O(1)$ amortized complexity; Section 4 illustrates the measurement methods and benchmarking used in the experiments and Section 5 presents the empirical results.

2. LADDER QUEUE

The LadderQ avoids the problems encountered by the SCQ by having four essential principles.

Firstly, LadderQ defers the sorting of events until absolutely necessary, that is, when some high priority events are close to being dequeued. The majority of arriving events are simply appended into buckets without sorting and hence only incur $O(1)$ cost.

Secondly, during enqueue operations, a resize operation encompasses only resizing the *Bottom* structure when it becomes too populated. The *Bottom* structure, made up of a sorted linked list, is the center of activity of the LadderQ and essentially determines the performance of the queue. However, all of LadderQ's other buckets do not get resized. Since the *Bottom* structure is limited to at most 50 events, this not only cuts down the costliness of the resize, it limits the complexity of the enqueue operations in *Bottom* which relies on linear search and also ensure that complexity requirements on *Bottom* is at most 50 operations (i.e., $O(1)$) irrespective of queue size and μ.

Thirdly, during dequeue operations, the LadderQ refines the bucketwidth on a bucket-by-bucket basis, reducing the cost of resize operations. Furthermore, this bucketwidth-refining methodology, which results in LadderQ having multiple bucketwidths, well handles event distributions which have widely varying inter-event time-gaps. In contrast, the SCQ adopts a "one size fits all" approach where only a single bucketwidth is used for the entire SCQ structure, resulting inevitably in costly resize operations.

Lastly, instead of unreliable and somewhat costly sampling heuristics, the LadderQ obtains good operating parameters dynamically, in accordance to the events in its structure, without sampling. This reduces further its management overheads on the whole.

2.1 Basic Structure of Ladder Queue

Figure 1 shows an example of the basic structure of a LadderQ. The name LadderQ arises from the semblance of the structure to a ladder with rungs. Basically, the structure consists of three tiers: a sorted linked list called *Bottom*; the middle layer, called *Ladder*, consisting of several rungs of buckets where each bucket may contain an unsorted linked list; and a simple unsorted linked list called *Top*. In the example of Figure 1, a LadderQ with three rungs of buckets – *Rung* [1], *Rung* [2] and *Rung* [3] is shown. The actual number of rungs may vary from distribution to distribution. Later, in Section 3, it will be demonstrated that the average number of rungs is bounded by a constant irrespective of N and as long as the mean jump parameter, that is, μ, of the distribution is finite and greater than zero, regardless of its variance.

In the literature, there is an interesting priority queue known as the Lazy Queue (LazyQ) [Rönngren et al. 1993] which bears a striking similarity in terms of structure as compared to the LadderQ which we describe in this article. The LazyQ also has three tiers. However, the similarity ends there. The mechanisms employed to trigger the transfer of events between the various tiers are more convoluted. It also requires the user to know *a priori* the value of numerous parameters for it to perform well. It also relies on the resize operation similar to the SCQ but on top of that it comes with various resize criteria. We only provide a brief mention of the LazyQ because it has been already shown that the LazyQ provides similar performance compared to the SCQ for most scenarios and worse for some scenarios [Rönngren and Ayani 1997]. As such, we have not included the LazyQ in our performance comparision. We have on the other hand empirically shown that the LadderQ significantly outperforms the SCQ in almost every scenario (see Section 5).

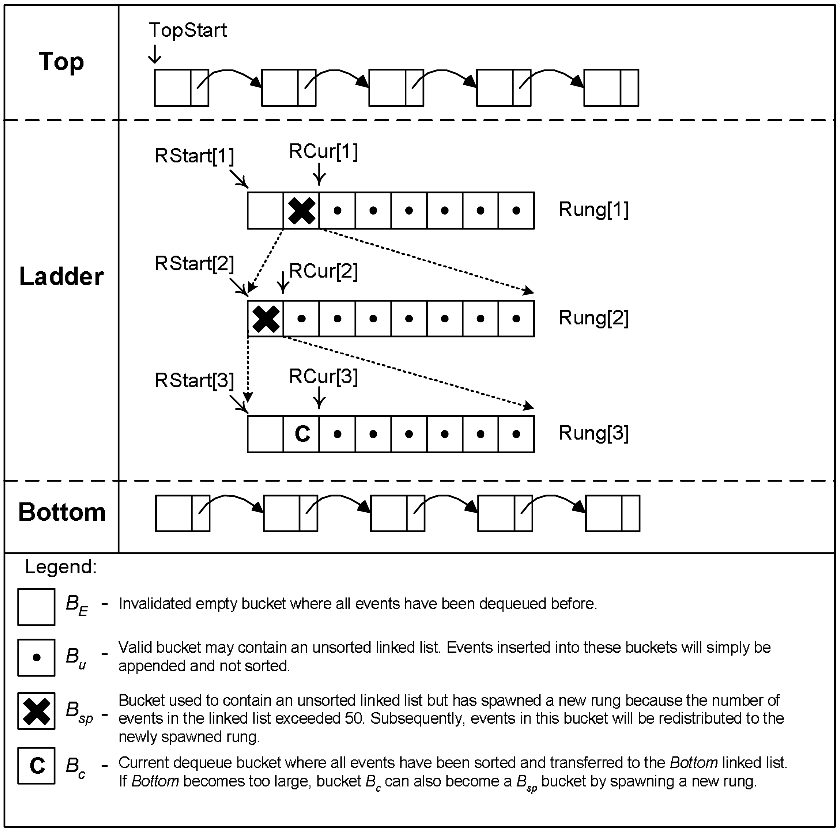


Fig. 1. Basic structure of Ladder Queue.

2.2 The Algorithm

Before we present a detailed description of the LadderQ algorithm, Table I describes the operating variables associated with this three-tier structure.

2.2.1 *Dequeue Operation.* Initially, *Top*, *Ladder* and *Bottom* are empty and all incoming events are inserted into *Top* without any sorting. *MaxTS*, *MinTS* and *NTop* are updated accordingly. When the first dequeue event is fired, events in *Top* are transferred to the first rung of *Ladder*, that is, *Rung* [1]. The bucketwidth of *Rung* [1] is obtained using

$$Bucketwidth\,[1] = \frac{MaxTS - MinTS}{NTop}, \tag{1}$$

where *MaxTs* is not equal to *MinTs*.[1] Upon obtaining the *Bucketwidth* [1], the following variables are updated: *RStart* [1] and *RCur* [1] are set = *MinTS*, and *TopStart* is set = *MaxTS* + *Bucketwidth* [1]. Thereafter, events can be

[1]Note that if *MaxTs* is equal to *MinTs*, it means all the events in *Top* have the same timestamp. In this article we consider the mean jump to be finite and positive. Thus, the likelihood of this occurring is extremely low. See Section 2.4 for the practical aspect of this occurrence.

Table I. Some Important Operating Variables Maintained in LadderQ

Tier	Variable	Definition/Purpose
Top	*MaxTS*	Maximum timestamp of all events in *Top*. Its value is updated as events are enqueued into *Top*.
	MinTS	Minimum timestamp of all events in *Top*. Its value is updated as events are enqueued into *Top*.
	NTop	Number of events in *Top*.
	TopStart	Minimum timestamp threshold of events which must be enqueued in *Top*.
Ladder	*Bucketwidth* $[x]$	*Bucketwidth* of *Rung* $[x]$.
	NB_c	Number of events in current dequeue bucket B_c.
	NBucket $[j, k]$	Number of events in *Bucket* [k] of *Rung* [j].
	NRung	Number of rungs currently in active use.
	RCur $[x]$	Starting timestamp threshold of the first valid bucket in *Rung* $[x]$ which subsequent dequeue operations will start. Minimum timestamp threshold of events which can be enqueued in *Rung* $[x]$.
	RStart $[x]$	Starting timestamp threshold of the first bucket in *Rung* [x]. Used for calculating the bucket-index when inserting an event in *Rung* $[x]$ of *Ladder*. See Eq. (2).
	THRES	If the number of events in a bucket or *Bottom* exceeds this threshold, then a spawning action would be initiated.
Bottom	*NBot*	Number of events in *Bottom*.

transferred to *Rung* [1] using

$$Bucket\_k = \left\lfloor \frac{TS - RStart\,[i]}{Bucketwidth\,[i]} \right\rfloor, \tag{2}$$

where $i = 1$. Equation (2) determines the index of bucket (*Bucket_k*) which an event is to be appended to the sublist of *Bucket_k*, unsorted. *TS* refers to the timestamp of an event being transferred. The rationale of using (1) to determine the bucketwidth is to assume that the inter-event time-gap in *Top* is uniformly distributed between *MaxTS* and *MinTS* so that there will be, on average, one event per bucket in *Rung* [1]. If this assumption of uniform distribution is not true, and that there are some buckets which contain too many events, then the spawning process of LadderQ will be activated to fix the problem. The spawning process of LadderQ is what creates the multiple rung structure of LadderQ as illustrated in Figure 1. This process is activated only when certain conditions are satisfied. More detailed description of the spawning process will be provided later.

Once the events have been transferred from *Top* to *Rung* [1], the buckets in *Rung* [1] will be traversed by the dequeue process to find the next nonempty bucket and this bucket will be designated as the current dequeue bucket B_c. The traversal of the buckets involves adding *Bucketwidth* [1] to *RCur* [1] for every bucket skipped. This ensures that buckets with timestamps less than *RCur* [1] are invalidated–that is, they should be empty of events.

Once the bucket B_c is found, the number of events in that bucket denoted by NB_c is compared with a predetermined threshold called *THRES*. The purpose of this threshold is to decide whether a spawning action should be initiated. If NB_c does not exceed *THRES*, they will be sorted and placed into the linked list

Bottom. Thereafter, the first event from *Bottom*, which is the event with the highest priority, will be returned.

However, if NB_c does exceed *THRES*, a spawning action will be initiated. B_c is converted into a B_{sp} (see Figure 1) and this process involves adding a new rung, that is, *Rung* [2], where the events from B_c are then transferred into. This spawning process is repeated until the sorted linked list *Bottom* is created and the highest priority event is dequeued.

For this spawning process, different variables are updated as compared to the transfer of events from *Top* to *Rung* [1]. In the example given in Figure 1, the variables *RStart* [2] and *RCur* [2] are set = *RCur* [1]. Subsequently, *RCur* [1] is incremented by *Bucketwidth* [1], and *Bucketwidth* [2] is set using

$$Bucketwidth\,[i+1] = \frac{Bucketwidth\,[i]}{THRES}. \quad (3)$$

Events held in the parent bucket in *Rung* [1] are then re-distributed into *Rung* [2] using the bucket-index procedure given in Eq. (2) with $i = 2$. The rationale of using Eq. (3) to obtain the bucketwidth of *Rung* [2] is based on the assumption that the events in the B_{sp} of *Rung* [1] are uniformly distributed within its bucketwidth. If not, the spawning process continues on with more rungs being created. The spawning process terminates when the number of events in the lowest rung does not exceed *THRES*. It is shown later that if the jump distribution has a finite μ, then the average number of rungs created bounded by a constant, irrespective of N.

The above describes all the possible operations that may result when a dequeue occurs. However, the majority of dequeue operations are much simpler once a previous dequeue operation has already created the sorted linked list *Bottom*. In such cases, since *Bottom* is a sorted linked list, there is a pointer that is always referencing to the first event in *Bottom*, that event is simply removed with negligible access times. When all the events in *Bottom* have been dequeued, the buckets of the lowest rung are traversed to find the next dequeue bucket B_c and the whole process is repeated again. When a child rung is dequeued of all events, the child structure is deleted and the dequeue operation shifts back to the parent rung.

Figure 2 illustrates an example of the dequeue operation just described. Initially, a series of enqueue operations insert events with timestamps 0.6, 0.5, 3.1, 3.05, 3.3, 3.4, 3.0 and 4.5 in that order into *Top*. On the first dequeue operation, events from *Top* are transferred to *Rung* [1] of *Ladder* as shown in the diagram, and the event with timestamp 0.5 is then dequeued. Subsequently, during the third dequeue operation, empty buckets are simply skipped and the sixth bucket is reached. For illustration purposes, it is assumed that having five events in the sixth bucket breaches the threshold *THRES*. The consequence of breaching *THRES* is that *Rung* [2] is spawned with a bucketwidth of 0.1. Next, the five events in the sixth bucket of *Rung* [1] are transferred to *Rung* [2]. Finally, the event with timestamp 3.0 is extracted at the third dequeue operation.

2.2.2 *Successive Dequeue Operations Creates LadderQ Epochs.* Finally, it should be noted that the LadderQ operates in epochs when successive dequeue

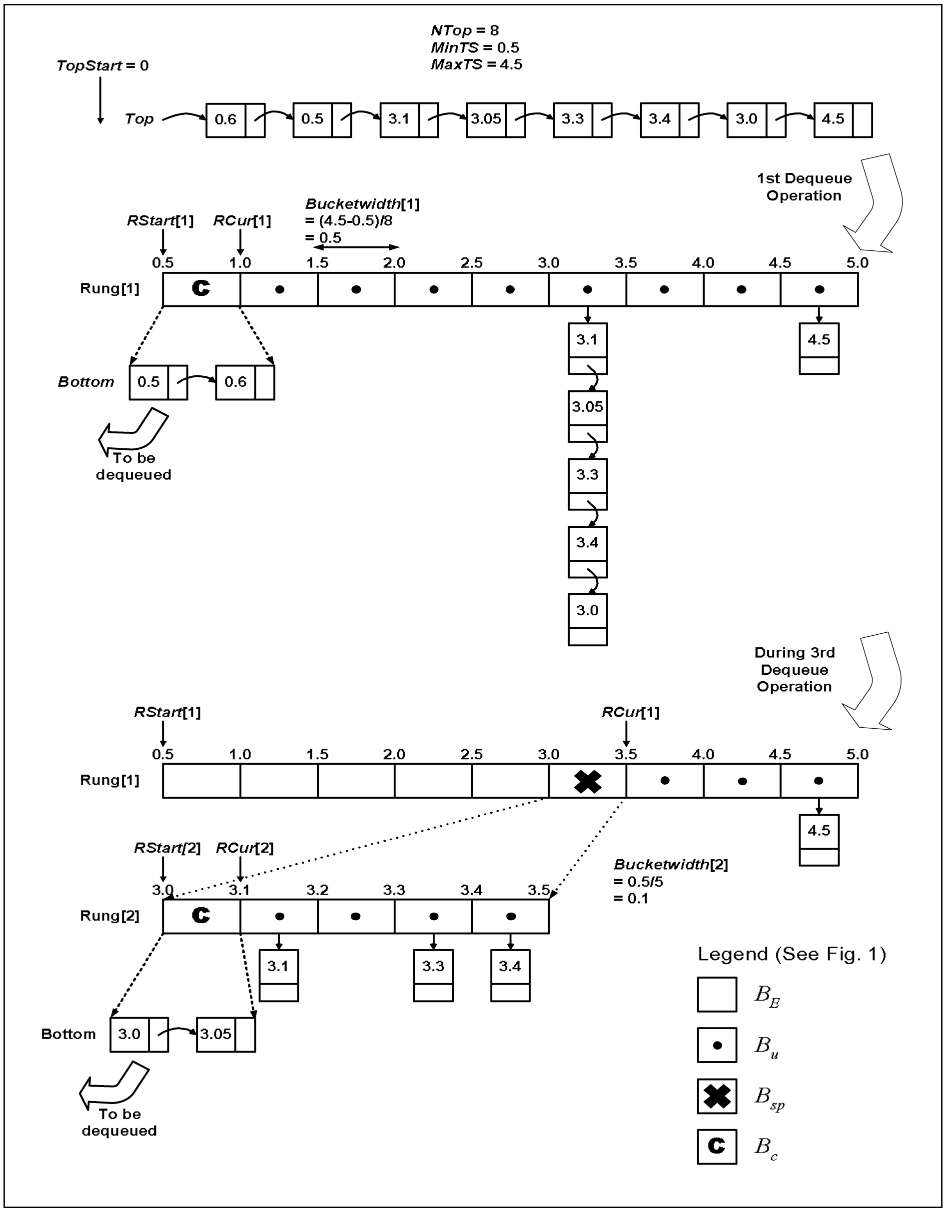


Fig. 2. An example illustrating the dequeue algorithm.

operations take place. *The start of a LadderQ epoch is marked by the creation of the Ladder and Bottom structure*. This occurs when all the events are held in *Top* (i.e., there is no *Ladder* structure and no *Bottom* structure currently) and the first dequeue operation is initiated (which creates the *Ladder* structure and the *Bottom* structure). The operating parameters of the epoch are determined using Eqs. (1), (2), (3). After the epoch is started, subsequent enqueue operations can result in events being enqueued in the *Top*, *Ladder* or *Bottom* structures. However, in a stable queue system where the number of events enqueued in the system is roughly equal to the number of events being dequeued, then each dequeue operation will advance the current time stamp, on average, by μ. This means that as the time-stamp advances, more and more buckets in the *Ladder* and *Bottom* structures will be dequeued of events and become empty while the *Top* structure will grow in size. Since the *Ladder* and *Bottom* structures have a finite number of buckets, there must exist a time where the *Ladder* and *Bottom* structures are exhausted of events and all future events are now found in *Top. When the Ladder* and *Bottom* structure are empty, then this marks the end of the epoch. The next epoch begins when equations (1), (2) and (3) are employed once again to derive the new operating parameters of the *Ladder* and *Bottom* structures. Notice that for the new epoch, Eqs. (1), (2) and (3) will be operating on parameters associated the current future events stored in *Top*. This means that LadderQ will always operate with a new set of operating parameters tailored for events in the new epoch. This is clearly desirable.

It is interesting to know that in a practical setting, some simulators such as Network Simulator v2 [Fall and Varadhan 2002] enqueues at the onset an event with a huge timestamp which indicates the end of a simulation task. This result in LadderQ having only one epoch and thus may slightly affect its performance. This is an isolated issue and can be easily solved by implementing two *Top* – Top_1 and Top_2, where Top_2 contains one or more huge timestamps. Top_1 then replaces the function of *Top* to allow multiple epochs.

2.2.3 *Enqueue Operation.* The enqueue operation in LadderQ is comparatively more straightforward than its dequeue operation. In each enqueue operation, the event timestamp is compared with the threshold of each level to find out where it is to be inserted. For each enqueue, the event timestamp (*TS*) is first compared with *TopStart*. If $TS \geqslant TopStart$, the event is appended at *Top*. Otherwise, *TS* is compared with thresholds $RCur[1], RCur[2], \ldots, RCur[NRung]$ to determine which rung the event should be in. Once $TS \geqslant RCur[i]$, the event is inserted into $Rung[i]$. The event is inserted into *Bottom* only if it does not belong to either *Top* or *Ladder*.

If the event is to be inserted into a $Rung[i]$ of the *Ladder*, the bucket index for insertion is determined using Eq. (2). As every bucket will contain a linked list that is not sorted in any order, the event is simply appended at the tail of the sublist. An insertion at *Bottom* will, on the other hand, require a sequential search through the list to determine the place of insertion as *Bottom* is always kept sorted. An enqueue into *Bottom* causing the *NBot* to reach *THRES* will activate a rung spawning process somewhat similar to that found in the dequeue operation. To describe this process briefly, assume that the number of rungs currently in use (*NRung*) is i. A new rung, $Rung[i+1]$, is created with $Bucketwidth[i+1]$ set using Eq. (3). Events that belonged to the *Bottom* list will then be redistributed into $Rung[i+1]$. The previous B_c in $Rung[i]$ then changes to B_{sp}. A new B_c is identified in $Rung[i+1]$ and events in B_c will then be sorted to form a new *Bottom* list.

Continuing from the dequeue example described in Figure 2, suppose two events with timestamps 5.6 and 3.2 are to be inserted. For the event with timestamp 5.6, it is simply appended into *Top* since $5.6 \geqslant TopStart$ $(= 5.0)$. For the other event, since its timestamp is less than $RCur[1]$ which is 3.5, it cannot be inserted in $Rung[1]$. Since event's timestamp is $3.2 \geqslant RCur[2]$ $(= 3.1)$ this event is to be enqueued in $Rung[2]$. Using Eq. (2) with $i = 2$, this event is then appended to the third bucket of $Rung[2]$.

2.3 Spawning vs Resize and Value of THRES

The rung spawning process in LadderQ can occur in both enqueue and dequeue operations. This is functionally different from the resize operation of the SCQ which involves the transfer of all events from an old queue to a new one with a different bucketwidth, which is an expensive procedure. However, the spawning operation in LadderQ only involves copying events from the current dequeue bucket B_c in *Ladder* or a single linked list in *Bottom* to a new rung. In short, spawning only affects a bucket or a linked list and not the entire PES structure. In comparison to the SCQ, this methodology is much more economical and efficient.

Recall that during an enqueue operation, it is possible for an event to be inserted in the *Bottom* linked list. If events are frequently enqueued in *Bottom*, it is possible for LadderQ to degenerate into a single linked list structure, thus degrading the performance of LadderQ severely. In such cases, it is therefore imperative for a new child rung to be spawned in *Ladder* and for *Bottom* to be recopied to it (B_c changes to B_{sp}). Hence, further enqueue operations in the lowest-most region will be further distributed into smaller buckets resulting in enqueues having $O(1)$ complexity instead of $O(n)$ complexity due to traversing a long sorted linked list. The other spawning operation is triggered during dequeue operations when the number of events in the current dequeue bucket B_c exceeds *THRES*. When this happens, a new child rung is created (B_c changes to B_{sp}), and events will be redistributed in the child rung.

Now, the value of *THRES* is independent of the event distribution and can be determined by comparing the average time required for sorting an event in *Bottom* and the average time required for spawning and copying *Bottom* events into a new child rung. A short piece of code that compares the time required per event under Linear Sort and Copy was written and run on an Intel Pentium 4 workstation. The simulation results in Figure 3 show that the threshold where Linear Sort intersects Copy is around 50. This means that during a dequeue operation, if the number of events is greater than 50, the time is better invested by copying the events to a new child rung. The value of 50 concurs with previous studies that linear sort is only efficient provided events in the list number less than 50 [Marin 1997].

2.4 Practical Aspects of LadderQ—Infinite Rung Spawning and Reusing *Ladder* Structure

The following presents some practical aspects of LadderQ for use in practical queue scenarios where some modification of the original LadderQ structure is required.

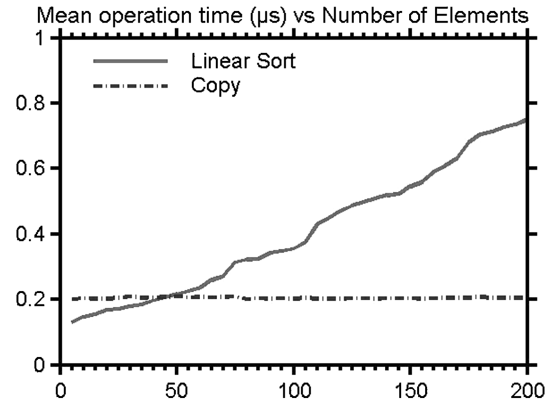


Fig. 3. Comparison of average time required per element for each operation.

First, the practical LadderQ, incorporates a maximum limit of eight rungs that can be spawned at any one time. This is to prevent infinite spawning of rungs in *Ladder*. In the extensive assortment of simulation scenarios considered in Section 5, the maximum number of rungs spawned did not exceed three. If there should be an occasion where there are already eight rungs, then events in the B_c (current dequeue bucket), associated with the eighth rung, are sorted to create *Bottom* even though the number of events in *Bottom* may exceed *THRES*. Infinite spawning of rungs can occur if the number of events exceeds *THRES* and the time stamps associated with all these events are all identical. This causes all the events to be always enqueued in one bucket irrespective of the number of rungs spawned. Even though scenarios with as much as 50 events having the same time stamp is rarely encountered, this safeguard eliminates the possibility of infinite spawning that may jeopardize the performance of LadderQ. For other more common priority increment distribution where the mean μ of the jump random variable is finite and greater than zero, then relevant theoretical justifications (see Section 3) will demonstrate that the average number of rungs spawned by the LadderQ is bounded by a constant.

Second, at the start of the simulation, the practical LadderQ is pre-initialized with five rungs (irrespective whether the distribution require less than five rungs), which is greater than the maximum number of rungs under most distributions (see Section 5). After each *epoch*, the rungs are not deleted but rather, they are reused for the subsequent *epochs*. The only difference is that the bucketwidth parameter of the rungs is different from epoch to epoch. By reusing the same rung structures, memory fragmentation is avoided and superior performance is obtained since the rungs are created only once. With the exception of *Rung*[1], the number of buckets in *Rung*[i], $i > 1$, is 50, that is, the *THRES* value. Hence pre-initializing *Rung*[2] to *Rung*[5] should be straightforward enough. In contrast, the number of buckets in *Rung*[1] is a variable which is only determined by equations (1) and (2) when a new epoch is started. To be able to reuse *Rung*[1] repeatedly for each epoch, we create more than enough buckets in *Rung*[1] when the first epoch starts. For example, if the actual number of buckets required in *Rung*[1] is M in the first epoch, then we create $2M$

buckets instead. If $2M$ buckets are insufficient for a later epoch, then at that epoch, a batch creation process is initiated again to double the amount of buckets in *Rung* [1]. Eventually, in a stable queue situation, the number of buckets in *Rung* [1] will progressively satisfy all the later epochs and thus do not require a batch creation of buckets. Hence, from epoch-to-epoch, if the number of buckets in *Rung* [1] is sufficient, the previous set of already created buckets can be reused.

With this cost-saving feature in the practical LadderQ, it is expected that the practical LadderQ will operate with more, or exactly enough, buckets than required in some epochs. Therefore, the practical LadderQ must also keep track of the total number of elements enqueued in each rung. When the number of elements parameter associated with a particular rung is zero, then the bucket-scanning process for that rung should cease, even though the current bucket is not the last bucket of that rung.

With the above modifications, the pseudo-code for the practical LadderQ can be better understood and is found in the Appendix.

3. THEORETICAL ANALYSIS OF LADDER QUEUE'S *O*(1) AVERAGE TIME COMPLEXITY

3.1 Scenario and Conditions for Theoretical Analysis and the 1-Epoch LadderQ

We now consider the theoretical performance of LadderQ. As mentioned earlier in Section 2.2.2, the algorithm of the LadderQ structure proceeds in epochs, whereby for each epoch, a new δ_1 is obtained according to Eq. (1) based on current events enqueued in *Top*. The LadderQ performs more favorably if there are more epochs in a simulation job since, firstly, each epoch redistributes the events in *Top* to a newly-created bucketwidth with parameters tailored for current events. Hence, parameters of past events do not affect the current operating parameters. Secondly, having more epochs suggests that as the simulation time progresses, more events will be enqueued in *Top* (which is an $O(1)$ cost per event since *Top* is an unsorted linked list) rather than *Bottom* and *Ladder*. This means that the *epoch population*,define to be the number of events in LadderQ excluding those events in *Top*, that is, the number of events in *Bottom* and *Ladder*, must decrease as timestamp increases. As the major complexity in LadderQ is not with its *Top* (as *Top* is always $O(1)$) structure, one can easily conclude that the major complexity of the LadderQ must therefore lie with its *Bottom* and*Ladder* structure. Hence we begin with the following Lemma:

LEMMA 3.1. *The complexity of the LadderQ cannot increase when the epoch population decreases.*

PROOF. The complexity of LadderQ lie primarily with its *Ladder* and *Bottom* structure as these structures contain sorting processes. When the epoch population decreases, this mean that the *Ladder* and *Bottom* structure manages less events compared to the time when the epoch was first created with maximum size. With smaller events to manage, less sorting is required and hence complexity cannot increase. □

It should be noted that the majority of practical simulation scenarios, including those with infinite variance in the jump variable, fit into the category where after the creation of an epoch, the epoch population becomes progressively smaller. Hence, the conventional LadderQ scenario is essentially a multi-epoch LadderQ where within the duration of the simulation, epochs arise and then die away eventually to give rise to another epoch. However, there are two (pathological) scenarios where the epoch population grows instead of decreases. The first scenario is an unstable queue situation where the number of events enqueued is always higher than the number of events dequeued. Hence it is possible for the epoch population to increase rather than decrease. Complexity analysis on queues implicitly require that there is some fixed N representing the target number of events in the queue for which a complexity measure is to be derived. A growing queue scenario is one where no target$N$ can be defined and hence bear no significance for further analysis. The other scenario where the epoch population increases is where the mean jump parameter μ is zero so that every event has the same timestamp. In this scenario, uncontrolled rung spawning may occur and is the reason behind the imposition of a maximum eight-rung limit to the practical LadderQ structure (see Section 2.4) and the relaxation of usual 50-element *Bottom* structure (i.e., sorted linked list) so that it can hold more than 50 elements. Even if such mystical scenarios are encountered, the LadderQ can also be easily adapted to yield $O(1)$ performance as follows: If there are already eight rungs and it is detected that all events are arriving with the same timestamp, then a special tail pointer for *Bottom* is initialized so that an enqueue process does not require event scanning beginning from the head of the queue. This makes LadderQ clearly an $O(1)$ structure, that is, all dequeue will be $O(1)$ using *Bottom*'s usual head pointer and all enqueue will also be $O(1)$ using *Bottom*'s special tail pointer.

We now return to the more conventional LadderQ scenarios where multiple epochs are encountered. Such scenarios have the property that the priority increment distribution has a finite mean jump parameter μ and the queue size grows to some well-defined value N, and then maintains at that level (i.e., the number of dequeues and enqueues are roughly the same) for some time. Consequently, the LadderQ structure will proceed in epochs which implicitly require that the current epoch population will decrease to zero at some stage in time (if the epoch population does not decrease, then the LadderQ cannot proceed in epochs). Also noted is that during each epoch, the bucketwidth parameter, δ_1, stays constant.

In the following theoretical analysis, we demonstrate that the LadderQ is $O(1)$ for conventional queue scenarios where the epoch population starts decreasing from the time it was created. However, in view of Lemma 3.1, it is also clear that the worst-case LadderQ complexity in a practical queue scenario is where the epoch population is at maximum and equal to N. In other words, it is reasonable to consider a 1-epoch LadderQ where all event activities (i.e., enqueue and dequeue activities) are assumed to only occur in *Bottom* or *Ladder* and the epoch population is always a constant equal to N. In addition, the 1-epoch LadderQ will now have a bucketwidth parameter δ_1 which remains constant throughout. The 1-epoch LadderQ also has some practical significance in that it represents the initial state of the multi-epoch LadderQ during the time when the epoch was first created with maximum events. It is also during this initial time following the epoch creation that the LadderQ experiences maximum time complexity. If this 1-epoch LadderQ is $O(1)$, then clearly, the multi-epoch LadderQ is also $O(1)$ (by virtue of Lemma 3.1). Hence, we state the following corollary, which is a result of Lemma 3.1:

COROLLARY 3.1. *The average time complexity of the multi-epoch LadderQ, that is, conventional LadderQ, is no larger than the average time complexity of the 1-epoch LadderQ.*

It should also be noted that the theoretical analysis do not consider the cost of rung creation since rung creation is only done once as explained in Section 2.4. As for the number of buckets in *Rung* [1]: if N is the target size of the queue, then based on (1) where we set $NTop = N$, the total number of buckets required in *Rung* [1] is just $N + 1$ (where the additional one bucket is to accommodate the maximum timestamp event) and this does not change any further during the epoch. It is noted that rung creation is just a fixed initial cost for each epoch irrespective whether we are considering a multi-epoch LadderQ or 1-epoch LadderQ. This fixed cost is $O(1)$ per event for each epoch since the cost of creating N buckets is a one-time cost when transferring N events from *Top* to *Ladder*.

Finally, an important issue for theoretical analysis is that we do not impose a maximum limit of eight rungs and the usual 50-element limit also applies to *Bottom* irrespective of the number rungs spawned. The case of practical LadderQ being limited to eight rungs as stated in Section 2.4 is only for a specific and obviously mystical queue situation where all the event timestamps are equal. However, as far as the mean jump parameter μ is finite and not zero, and the event size does not grow infinitely, then a maximum rung limit has no theoretical basis for its existence. This will be clear in the following sections.

3.2 Complexity of 1-epoch LadderQ

The complexity of tree-based priority queues (e.g., Splay Tree [Sleator and Tarjan [1985]) are often gauged by considering the average height of the growing tree structure. Similarly, the complexity of the 1-epoch LadderQ is closely related to the average number of rungs that are spawned, as will be seen later. We begin with the following proposition:

PROPOSITION 3.1. *The* 1*-epoch LadderQ is* $O(1)$ *if the average number of rungs for a priority event distribution is bounded by a constant.*

Justification. The proposition can be justified by considering the cost of a dequeue operation and the cost of an enqueue operation in a 1-epoch LadderQ as the number of events N increases. If the costs of these two bread- and- butter operations in the 1-epoch LadderQ are O(1), then the structure is also O(1). We consider these two basic costs and analyze them in terms their fixed cost and variable cost component.

Dequeue Cost. A fixed cost of dequeue is incurred when there is already a *Bottom* list. In this situation, since there is always a pointer that points to the first event in the sorted *Bottom* list, the fixed cost of dequeue is to reference this pointer, extract the first element and then increment the pointer to the next element. This is $O(1)$ irrespective of N. However, a dequeue will incur a variable cost when there is no *Bottom* list. In that case, several rungs may be spawned. In each spawned rung, there will be *THRES* number of buckets. If we consider the worst-case, the most number of buckets in *Rung* [1] that requires spawning is $(N+1)/THRES$ buckets, since there are at most $N+1$ buckets in *Rung* [1] (where the additional one bucket is to accommodate the maximum timestamp event). Each bucket spawned generates *Rung* [2]. Since each spawned *Rung* [2] contains only *THRES* number of buckets, there will be at most $(N+1)/THRES$ * $THRES = (N+1)$ total number of buckets to be traversed for *Rung* [2]. And the analysis is the same for subsequent rungs. The total cost (worst-case) for transferring N events during dequeue operations from *Top* to *Rung* [1] is N; from *Rung* [1] to *Rung* [c] is $(c-1)(N+1)$ and from *Rung* [c] to *Bottom* is $N+1$. This gives a total worst-case transfer cost (from *Top* to *Bottom*) during all dequeues as $N + c(N+1) = O(N(c+1))$, where c is the number of rungs spawned. Therefore if the average number of rungs spawned is bounded by a constant irrespective of N, then a dequeue operation involving rung spawning is $O(1)$ amortized (see Corollary 3.4).

Enqueue Cost. An $O(1)$ fixed cost of enqueue is incurred by calculating the bucket index for inserting an event as well as appending the event into an unsorted bucket. If the event is to be inserted into *Bottom* using Linear Sort, the search cost is also bounded as the number of events in *Bottom* is always limited to 50. However, an enqueue can incur a variable cost if rungs are required to be traversed in order to find the appropriate insertion level. Therefore, if the average number of rungs is bounded by a constant irrespective of N, this variable enqueue cost is also bounded.

3.3 Similarity of 1-Epoch LadderQ to the UCQ

To prove that the average number of rungs in 1-epoch LadderQ is bounded, we adopt a static Hold scenario as used by Erickson et al. [2000] to show that a UCQ has $O(1)$ amortized complexity. It is under the static Hold scenario, where the number of events N is kept a constant, will it be possible that a Markov chain model be able to provide an analysis of the UCQ structure. The UCQ model considered in that article assumed that the N events are all stored within the same year of the UCQ's bucket structure. These N events are also assumed to be created by a priority increment distribution where the mean of the jump random variable is some constant μ, which is finite and positive, regardless of its variance. This UCQ scenario has an uncanny resemblance to the first rung of the *Ladder* portion of Figure 1. We can apply the same scenario of interest to the 1-epoch LadderQ as was applied to the UCQ. Initially, N events are generated and placed into the 1-epoch LadderQ's *Top* list. Once N events have been queued in *Top*, the first dequeue is generated and this causes *Rung* [1] in the *Ladder* portion to be created to store the N events. The creation of *Rung* [1] would place all N events within one calendar year of *Rung* [1]. If child rungs are spawned, then we consider all events stored in child rungs to belong to the spawning parent bucket. Hence, the distribution of events in *Rung* [1] of the *Ladder* portion is identical to the UCQ scenario considered in Erickson et al. [2000].

3.4 Useful Lemmas Applicable to the 1-Epoch LadderQ

Several lemmas, which are to be used later for showing the average number of rungs in a 1-epoch LadderQ is bounded by a constant, are now presented. For convenience, we define the following variables:

B_i represents the random variable that the bucket B_c or B_{sp} in *Rung* [i] contains a certain number of events ranging from 0 to N.

δ_i represents the bucketwidth of *Rung* [i]. Based on (3) and for *THRES* $= 50$, we note that $\delta_{i+1} = \delta_i/50 = \delta_1/50^i$. Note that the analysis is for the 1-epoch LadderQ where δ_1 is a constant throughout.

μ is the finite mean of the jump random variable that defines the priority increment distribution of the N events. The jump random variable has a cumulative density function denoted by $F(x)$.

$q_{i,j}$ is used to denote the limiting probability that a bucket in *Rung* [i] has exactly j events enqueued inside.

LEMMA 3.2. *For $N \geq 2$ and all $\delta_1 > 0$, the N events are distributed amongst the one year, UCQ-like, first rung structure of the* 1*-epoch LadderQ according to*

$$P(B_1 = 0) = q_{1,0} = \frac{\mu}{\mu + N\delta_1}, \tag{4}$$

and for $j = 1, 2, \ldots, N$, we have

$$q_{1,0}B(j) \leq P(B_1 = j) = q_{1,j} \leq \frac{q_{1,0}B(j)}{1 - F(\delta_1)}, \tag{5}$$

where $B(j)$ is the tail probability of a binomial distribution for N trials with "success" parameter $p(\delta_1)$:

$$B(j) = \sum_{k=j}^{N} \binom{N}{k} p^k (1-p)^{N-k}, \tag{6}$$

$$p(\delta_1) = \frac{1}{\mu} \int_0^{\delta_1} [1 - F(x)]dx. \tag{7}$$

Remark. Lemma 3.2 provides a mathematical description to the UCQ distribution under a Hold scenario. The relevant proof is presented in Erickson et al. [2000] using a Markov chain model.

The next Lemma presents results on the probability distribution of events in the child rungs. For this purpose, it can be assumed, without loss of generality, that each bucket in *Rung* [1] will spawn m number of buckets as shown in Figure 4. For the 1-epoch LadderQ, $m = 50$. However, the proof is also applicable for all $m > 1$. An event enqueued in *Rung* [1] can be seen to be virtually enqueued in *Rung* [2]. This virtual enqueue process can also be applied to higher

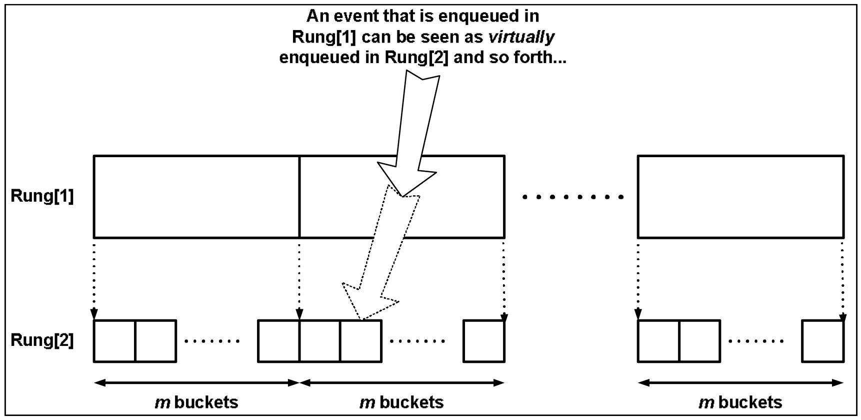


Fig. 4. Events in a parent bucket are virtually enqueued amongst m child buckets.

order rungs where virtual events in a parent bucket are seen to be virtually enqueued amongst m number of buckets in the child rung.

With the virtual enqueue process in place, it is clear that each child *Rung* $[i]$ can be considered to be a one year UCQ with bucketwidth $\frac{\delta_1}{m^{i-1}}$. Hence, Lemma 3.1 is also applicable on the successive child rungs so that the following probability expression is valid:

$$P(B_i = 0) = q_{i,0} = \frac{\mu}{\mu + N\frac{\delta_1}{m^{i-1}}}. \tag{8}$$

We now present Lemma 3.3 as follows:

LEMMA 3.3. *As more levels of child rungs are spawned, the probability that a bucket in the lowest child rung, denoted to be the Lth rung, has no element will asymptotically approach unity, that is,* $q_{1,0} < q_{2,0} < K < q_{L-1,0} < q_{L,0} < K \to 1$.

PROOF. The proof is seen in (8) where we let i increase to a large value. □

Remark. Lemma 3.3 is also rather intuitive since as more child rungs are spawned, the more the events are spread out amongst the child rungs and thus the greater the likelihood that there will be no element in any particular bucket belonging to the lowest rung. This lemma also suggests that the number of child rungs spawned ought to be bounded.

LEMMA 3.4. *Let L represent the random variable that counts the number of rungs spawned in a 1-epoch LadderQ. Then for* $n > 1$, $P(L = n) < (1 - q_{n-1,0})$.

PROOF. We provide proofs for $P(L = 2)$ and $P(L = 3)$ and then apply induction to obtain the desired result for $P(L = n)$. It is noted that for 1-epoch LadderQ, the new child rung is spawned when the number of events in a parent bucket has reached 50. Hence, we note that:

$$\begin{aligned}
P(L = 1) &= P(B_1 < 50)\\
P(L = 2) &= P(B_2 < 50, B_1 \geq 50)\\
&\leq P(B_1 \geq 50) = 1 - P(B_1 < 50)\\
&< 1 - q_{1,0}.
\end{aligned}$$

Similarly,

$$\begin{aligned} P(L=3) &= P(B_3 < 50, B_2 \geq 50, B_1 \geq 50) \\ &\leq P(B_2 \geq 50) = 1 - P(B_2 < 50) \\ &< 1 - q_{2,0}. \end{aligned}$$

Similarly, bounds for $P(L = 4)$ and higher can be derived using similar techniques and hence the proof is complete. □

3.5 Theorems for the 1-Epoch LadderQ's $O(1)$ Amortized Complexity

We present in this section several theorems and corollaries for the 1-epoch LadderQ. We begin with Theorem 3.1 as follows:

THEOREM 3.1. *The average number of rungs in a* 1*-epoch LadderQ is bounded by a constant provided the bucketwidth* δ *is of* $O(1/N)$.

PROOF. The average number of rungs for a 1-epoch LadderQ, containing N events, is given by:

$$\begin{aligned} EN[L] &= \sum_{j=1}^{\infty} jP\,(L=j) \\ &= 1.P(L=1) + 2.P(L=2) + 3.P(L=3) + 4.P(L=4) + L \\ &< 1 + 2(1-q_{1,0}) + 3(1-q_{2,0}) + 4(1-q_{3,0}) + L \text{ (see Lemma 3.3)} \\ &= 1 + 2\left(\frac{N\delta_1}{\mu + N\delta_1}\right) + 3\left(\frac{N\delta_1}{m\mu + N\delta_1}\right) + 4\left(\frac{N\delta_1}{m^2\mu + N\delta_1}\right) + L \\ &\leq 1 + \frac{N\delta_1}{\mu}\left(1 + \frac{2}{m} + \frac{3}{m^2} + \cdots\right) + \frac{N\delta_1}{\mu}\left(1 + \frac{1}{m} + \frac{1}{m^2} + \cdots\right) \\ &= 1 + \frac{N\delta_1}{\mu}\left(\frac{1}{\left(1-m^{-1}\right)^2} + \frac{1}{1-m^{-1}}\right) = 1 + \frac{N\delta_1}{\mu}\left(\frac{2m^2 - m}{m^2 - 2m + 1}\right) < \infty \quad (9) \end{aligned}$$

The last expression in (9) is bounded by a constant as long as the bucketwidth δ_1 of *Rung* [1] is held at $O(1/N)$. □

The next theorem, that is, Theorem 3.2, provides an alternative proof to demonstrate that the average number of rungs in the 1-epoch LadderQ is bounded by a constant. While Theorem 3.1 has already provided an explicit bound (see (Eq. 9)) for the average number of rungs, a constraint is required on the 1-epoch LadderQ's δ_1 parameter, that is, $\delta_1 \sim O(1/N)$.

Theorem 3.2, on the other hand, does not require any constraint on the δ_1 parameter but will not provide an explicit bound value as Theorem 3.1 did.

THEOREM 3.2. *The average number of rungs in a* 1*-epoch LadderQ is bounded by a constant regardless of the Rung* [1] *bucketwidth* δ_1.

PROOF. We begin with the usual expression for obtaining the average number of rungs for a total of N events in the 1-epoch LadderQ:

$$\begin{aligned}
E_N[L] &= \sum_{j=1}^{\infty} jP(L=j) \\
&= 1.P(L=1) + 2.P(L=2) + 3.P(L=3) + 4.P(L=4) + L \\
&< 1 + 2(1-q_{1,0}) + 3(1-q_{2,0}) + 4(1-q_{3,0}) + L \text{ (see Lemma 3.3)} \cdots \\
&\quad + T_j + T_{j+1} + \cdots,
\end{aligned}$$

where T_j and T_{j+1} represents the higher order terms of the series. The above sum of series is bounded as long as the ratio test T_{j+1}/T_j is less than 1. Thus,

$$\lim_{j\to\infty} \frac{T_{j+1}}{T_j} = \lim_{j\to\infty} \left(\frac{j+1}{j}\right) \frac{(1-q_{j+1,0})}{(1-q_{j,0})} \quad = \lim_{j\to\infty} \left(\frac{m^{j-1}\mu + N\delta_1}{m^j\mu + N\delta_1}\right) = \frac{1}{m} < 1.$$

Since the ratio test evaluates to a value that is less than unity, the average number of rungs must converge to some constant less than infinity. □

COROLLARY 3.2. *The* 1*-epoch LadderQ has* $O(1)$ *average time complexity.*

PROOF. The proof is obtained by combining Proposition 3.1 with either Theorem 3.1 or Theorem 3.2. □

COROLLARY 3.3. *The* 1*-epoch LadderQ has O(N) total memory usage.*

PROOF. As shown in Eq. (1), *Ladder's* first rung requires $N+1$ ($\approx N$) buckets on a transfer of events from *Top* and thus the first rung has an $O(N)$ memory usage. Each subsequent child rung requires $O(\mathit{THRES})$ memory space. Since the average number of rungs is bounded (See Theorem 3.1 or Theorem 3.2), the 1-epoch LadderQ's memory consumption is therefore bounded by $O(N)$. □

COROLLARY 3.4. *The average amortized dequeue cost incurred when events are transferred from Top to the multirung Ladder structure and then to the Bottom structure is* $O(1)$*.*

PROOF. Assume that the average number of rungs spawned is C. Therefore, the worst-case total cost (note: total cost is not the amortized cost) incurred is given as $O(N(C+1))$, where all the events in *Top* traversed C number of rungs before reaching *Bottom* (see Proposition 3.1). Hence, the cost incurred per event is $O(C+1) = O(1)$ since C is some constant independent of N, which is given in Theorems 3.1 and 3.2. □

3.6 Theorems for the Conventional LadderQ's $O(1)$ Amortized Complexity

For formality, we now have

COROLLARY 3.5. *The conventional (multi-epoch) LadderQ is, theoretically, an O(1) priority event queue structure.*

PROOF. Combine the results of Corollary 3.1 and Corollary 3.2. □

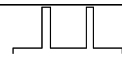
LadderQ, with its theoretical $O(1)$ amortized complexity, is clearly and significantly more robust than the current $O(1)$ priority queue structure proposed. It should be noted that the UCQ considered by Erickson et al. [2000] is $O(1)$ only provided the bucketwidth can be kept at $O(1/N)$. Therefore, for the UCQ to maintain $O(1)$ in a dynamic queue situation where N varies, costly bucketwidth resizes must be initiated to adjust the bucketwidth and hence the UCQ has never been widely used as the priority queue structure for practical simulators. In comparison with the more widely implemented SCQ, the LadderQ is also more superior. First, in the area of amortized complexity, the SCQ can at best be described as having *expected* $O(1)$ complexity, meaning that there is no known theoretical proof to show $O(1)$ except through a number of simulation examples. In fact, simulations studies conducted on the SCQ has shown that for certain scenarios where the priority increment is highly unstable, the SCQ exhibit $O(N)$ characteristics either due to under or over triggering of resize operations. To demonstrate the usefulness and superiority of the LadderQ for practical implementation, Sections 4 and 5 provide simulation studies on the performance of the LadderQ in comparison with previously proposed priority queue structures. It should be noted that most of the scenarios chosen for the simulation studies are those that have been previously proposed by well-known researchers on priority queues. Other scenarios presented have been suggested by the reviewers to incorporate more stringent tests. In fact, the LadderQ does not require any special scenarios to show its superiority since it already has the distinction being $O(1)$ theoretically, irrespective of the bucketwidth parameter (and hence N). We note that in the numerical studies presented in Section 5, the maximum number of rungs spawned never exceeded three levels.

4. PERFORMANCE MEASUREMENT TECHNIQUES

The performance of priority queues are often measured by the average access time to enqueue or dequeue an event under different load conditions. The parameters to be varied for each queue are: the access pattern, the priority distribution and the queue size. The access pattern models that have been proposed either emulate the steady-state or the transient phase of a typical simulation; Classic Hold model [Jones 1986] and Up/Down model [Rönngren et al. 1993] respectively.

The priority increment distributions used for the benchmarking of priority queue structures are found in Table II, where rand() returns a random number [Park and Miller 1988] in the interval [0,1]. The Camel(x, y) distribution [Rönngren et al. 1993] represents a 2-hump heavily skewed distribution with $x\%$ of its mass concentrated in the two humps and the duration of the two humps is $y\%$ of the total interval. The Change(A, B, x) distribution [Rönngren et al. 1993] was also used to test the sensitivity of the SCQ when exposed to drastic changes in priority increment distribution. The compound distribution Change(A,B,x) interleaves two different priority increment distributions A and B together. Initially, x priority increments are drawn from A followed by another x priority increments drawn from B and so on. Change distributions can be used to model simulations where the priority increment distributions vary

Table II. Priority Increment Distributions

	Distribution	Expression to compute random number
1	Exponential(1)	**if** `rand() == 0` **then** `infinity` **else** `−ln(rand())`
2	Bimodal	9.95238*`rand()` + **if** `rand()<0.1` **then** `9.5238` **else** `0`
3	Camel(0,1000,0.001,0.999)	 `See [Rönngren et al. 1993]`
4	Change(exp(1),Triangular (90000,100000),2000)	`Counter++, Counter %= 4000` **if** `(Counter<2000)` **then** `Exponential (1)` **else** `90000 + Triangular()*10000`
5	Pareto (α)	`pow(1/(1-rand()),1/α)`

significantly over different time periods, for example, battlefield simulations. The Pareto(C,D) is a heavy-tailed distribution and is an excellent model for event timestamps with high variance. C is the shape parameter and D is the scale parameter. D affects the range where the random number generated is $\geq D$. In this article, we set $D = 1$. C affects the mean and variance; Type 1– Infinite mean & infinite variance: $0 < C \leq 1$, Type 2 – Finite mean & infinite variance: $1 < C \leq 2$ and Type 3 – Finite mean & finite variance: $C > 2$. We have included the first two types, Pareto(1) and Pareto(1.5), since their mean and/or variance are different from the other distributions included in this article.

The hardware platform used for the experiments was a 2.4-GHz Intel Pentium 4. It is equipped with 1GB of shared RAM. The operating system on this hardware is Linux Mandrake 9.0 with kernel 2.4.19. Background processes that might affect the benchmark results were kept at a minimum. A microsecond resolution timer was used. Two empirical tests were conducted to verify that no items in the priority queues were gained or lost and that successive dequeues removed events in stable time-order.

In addition, the experiments were performed with the required memory for each priority queue being pre-allocated. This was to eliminate the underlying memory management system which might affect the results. This is a good practice in actual DES as it prevents memory fragmentation when creating new events and deleting the serviced events. This method of pre-allocating memory would also enhance the performance of the DES. The method of pre-allocation could be made dynamic by an initial pre-allocation and subsequently, an allocation of memory on demand methodology could be employed. All code was written in the C programming language with all recursive procedure calls and the like being eliminated. Loop overhead time and the time taken for random numbers generated were removed by factoring out the time required for running a dummy loop.

5. EMPIRICAL RESULTS

Figures 5 and 6 demonstrate the performance of the priority queues for Classic Hold and Up/Down experiments respectively and for increasing queue sizes with various priority increment distributions. The results shown are the average values from five identical runs. The obvious "knee" phenomenon at queue size 10,000 is due to declining cache performance, which has also been reported by Rönngren and Ayani [1997].

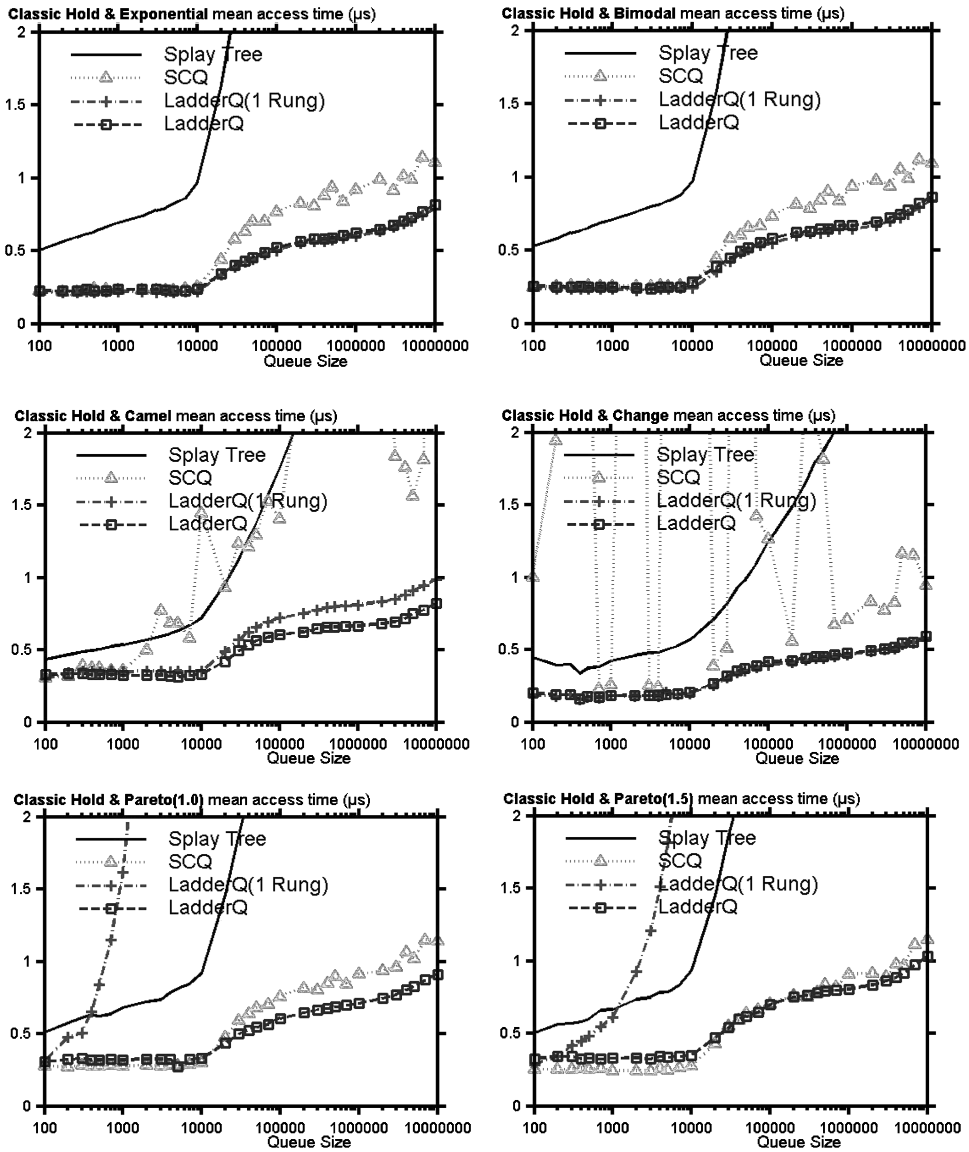


Fig. 5. Mean access time for Classic Hold experiments under different distributions.

The LadderQ performs very stably under all distributions with excellent performance for all queue sizes tested, that is, 100 to 10 million. To demonstrate the importance of the *Ladder* structure within LadderQ, we have included in the experiments a single rung LadderQ denoted as LadderQ(1 Rung). Figure 5 shows that LadderQ(1 Rung) is $O(n)$ for Pareto, a heavy-tailed distribution and Figure 6 clearly reveals the weakness of LadderQ(1 Rung) under Camel, Change and Pareto distributions where many events ended up in some few buckets, leading to a long *Bottom* linked list. Because the LadderQ(1 Rung)

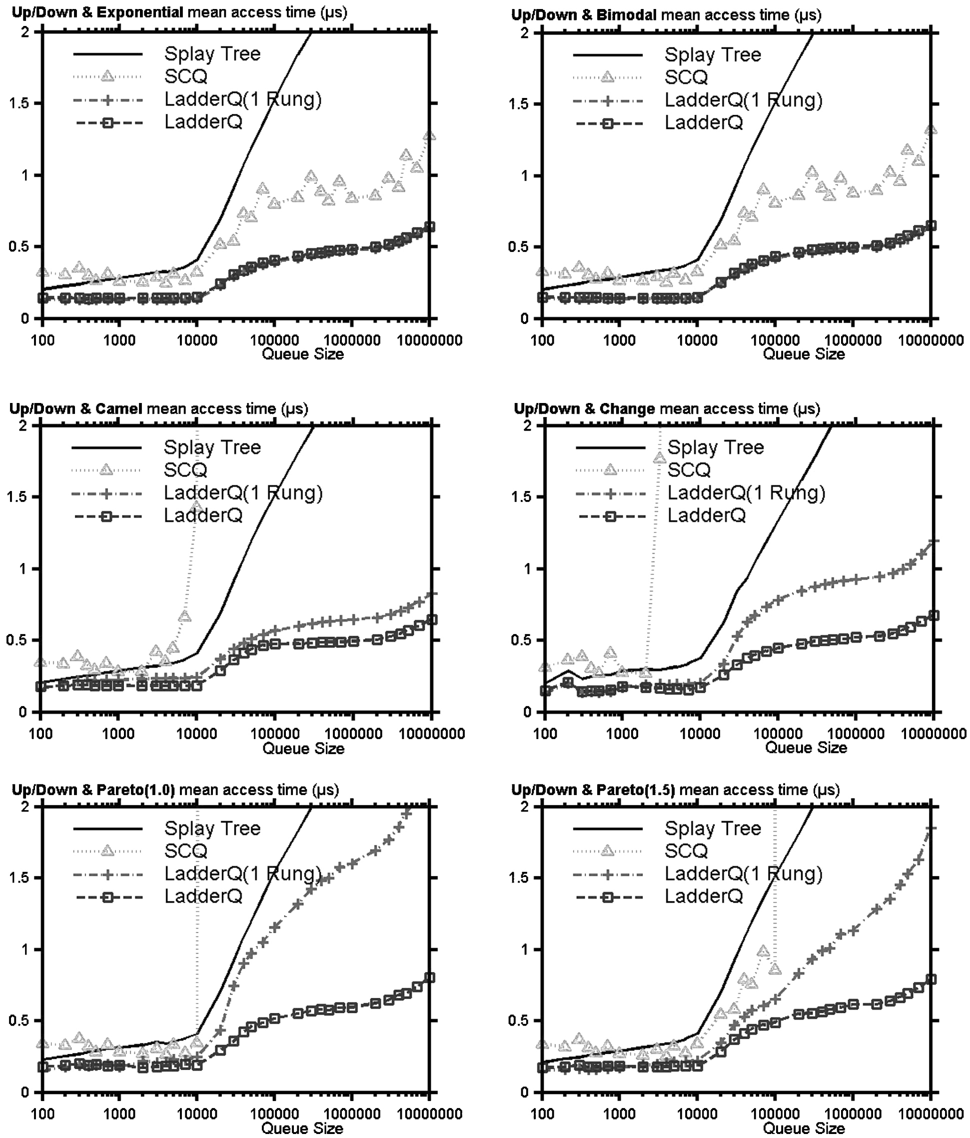


Fig. 6. Mean access time for Up/Down experiments under different distributions.

does not resize or spawn, the bucketwidth stays fixed and the number of events in *Bottom* cannot be reduced to manageable quantity. This clearly is not a high-quality methodology and results in a poor overall performance for skewed and heavy-tailed distributions. These observations therefore speak for the importance of *Ladder* structure for the LadderQ to achieve its $O(1)$ characteristic.

The SCQ, a popular PES structure for DES, performs well for distributions where the mean of the jump remains constant under the Classic Hold experiments shown in Figure 5. However, SCQ is extremely sensitive to the skewed distributions Camel and Change, where the mean of the jump varies.

Table III. Maximum Number of Rungs Utilized in Classic Hold and Up/Down Experiments

Distribution	Under Hold	Under Up/Down
Exponential (1)	2	1
Bimodal	1	1
Camel (0,1000,0.001,0.999)	2	3
Change (exp(1),Triangular (90000,100000),2000)	3	3
Pareto (1.0)	3	3
Pareto (1.5)	2	3

For Figure 6, where the queue size fluctuates during each experiment, the SCQ is evidently not a suitable PES candidate for simulation jobs of this nature. For exponential and bimodal distributions, the SCQ is closed to 100% slower than the LadderQ. For skewed and heavy-tailed distributions, the SCQ is very unstable at $O(n)$. The reasons for the behavior of the SCQ plots have already been elucidated in points (a) to (d) of Section 1 (i.e., introduction section).

The Splay Tree is an efficient tree-based priority queue. Though it is only at best $O(\log(n))$, it is included for two important reasons:

To demonstrate that cache effects do exist. The Splay Tree has been theoretically proven to perform at $O(\log(n))$ worst-case amortized complexity. However, it also exhibits the "knee" effects at 10,000-queue size.

To provide a comparison between an $O(\log(n))$ and $O(1)$ priority queues. Because of cache effect, the performance of $O(1)$ priority queues may not be visually obvious. The splay tree plots provide the necessary comparison, which undoubtedly affirms the superior performance of the $O(1)$ LadderQ.

Table III shows a summary of the maximum number of rungs spawned in the experiments. The number of rungs does not exceed three under all the distributions and queue sizes.

5.1 Experiments on an Intel Pentium 4 2.4 GHz without Cache—Effect of Bucketwidth on the Performance of LadderQ and the Actual Algorithmic Complexity

All the prior simulation results for the LadderQ demonstrate knee effects due to declining cache performance. Hence, it is certainly not visible from Figure 5 and Figure 6 that the LadderQ is truly $O(1)$. In this section, we present numerical simulations of various priority queues under the same Pentium 4 2.4-GHz platform but with cache disabled. Having the cache disabled ensure that the hardware only accesses a standard memory type with the same access speed irrespective of the size of N. Hence, a true $O(1)$ priority queue like LadderQ, if simulated without cache, would demonstrate an average amortized time complexity that is a constant straight line across all queue sizes. Our first experiment without cache is to demonstrate that the bucketwidth δ_1 does not affect the $O(1)$ characteristic of the LadderQ as elucidated in Theorem 3.2. We note in Figure 7 that when δ_1 is varied from $O(1/100N)$ to $O(100/N)$, the mean access time remains relatively constant across all queue sizes. The LadderQ with a $O(1/100N)$ bucketwidth results in slightly poorer performance (but still retains the $O(1)$ constant mean access time characteristic). This is due to the additional cost of skipping more empty buckets since for δ_1 to be $O(1/100N)$, there are 100

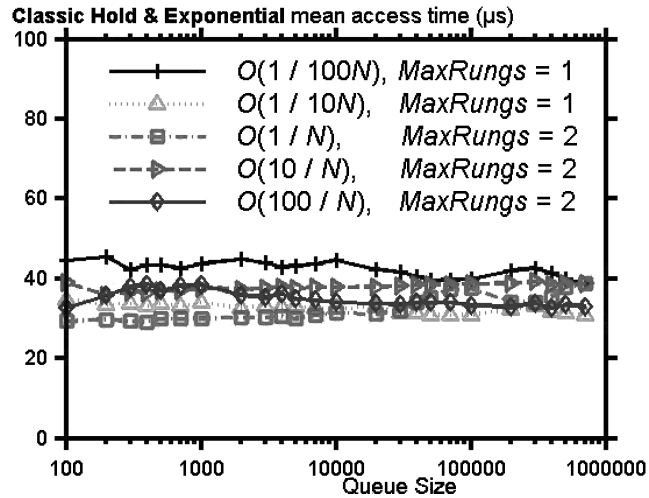


Fig. 7. Mean access time of LadderQ (with widely-varying bucketwidth) for Classic Hold experiments and exponential distribution. Maximum number of rungs (*MaxRungs*) used during the experiments are also shown.

times more buckets created in *Rung* [1]. In distinct contrast, when the bucketwidth of the UCQ varies by a few magnitudes, the mean access time varies by several factors (see Figure 1 in Erickson et al. [2000]).

Figure 8 demonstrates the reliable $O(1)$ characteristics of LadderQ under a wide variety of priority increment distributions found in the landmark survey paper by Rönngren and Ayani [1997]. In addition we have included the Pareto(1) and Pareto(1.5) distributions which allow us to test the *jump* parameter with infinite mean and infinite variance, and finite mean and infinite variance respectively. It is stressed that the constant line $O(1)$ characteristic is only visible if the cache is disabled. If cache is enabled, then the plots in Figure 8 would also demonstrate knee effects similar to Figure 5 and Figure 6.

5.2 Performance Evaluation via SWAN

To determine the performance of LadderQ in actual discrete event simulations, the LadderQ, Splay Tree and the SCQ have been implemented as the PES structures available in the SWAN (Simulator Without A Name) simulator. A simple $M/M/1$ queuing system was created in SWAN and simulated for 10,000 simulation seconds. In this network topology, each source node generates a network packet where the inter-packet generation rate is exponentially distributed. The source nodes, which vary from 8 to 399,998, generate packets which are multiplexed into a single server that services the packets to a single destination node. Including the server and the destination nodes, the total number of nodes in the network topology varies from 10 to 400,000. The service times for the packets are also exponentially distributed. The results obtained are shown in Figure 9.

Each discrete event simulator operates uniquely. For SWAN, an event in the PES corresponds to a node in the network topology. At the onset, if the number of nodes is set to be N, SWAN initializes the PES structure with N

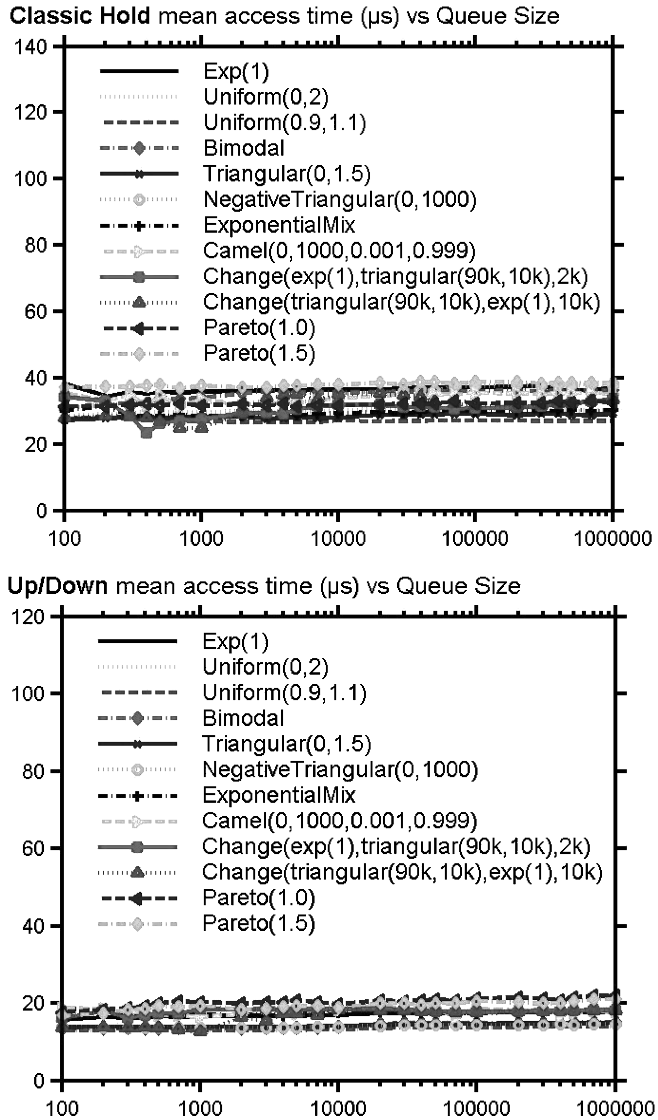


Fig. 8. Mean access time of LadderQ for Classic Hold and Up/Down under different distributions.

events with each holding a zero timestamp. Thereafter, the process interactions between the nodes determine the dequeue and enqueue of subsequent events with the number of events in the PES structure being held constant throughout the simulation at N. Thus, the mechanism of SWAN essentially corresponds to the Classic Hold model where the number of events in the PES structure is a constant and is equal to the number of nodes in the network topology. The x-axis in Figure 9 can thus be relabeled as "Number of nodes."

The benchmarks in this section were carried out on the Intel Pentium 4 workstation. The metric measured was the overall run-time, which include the

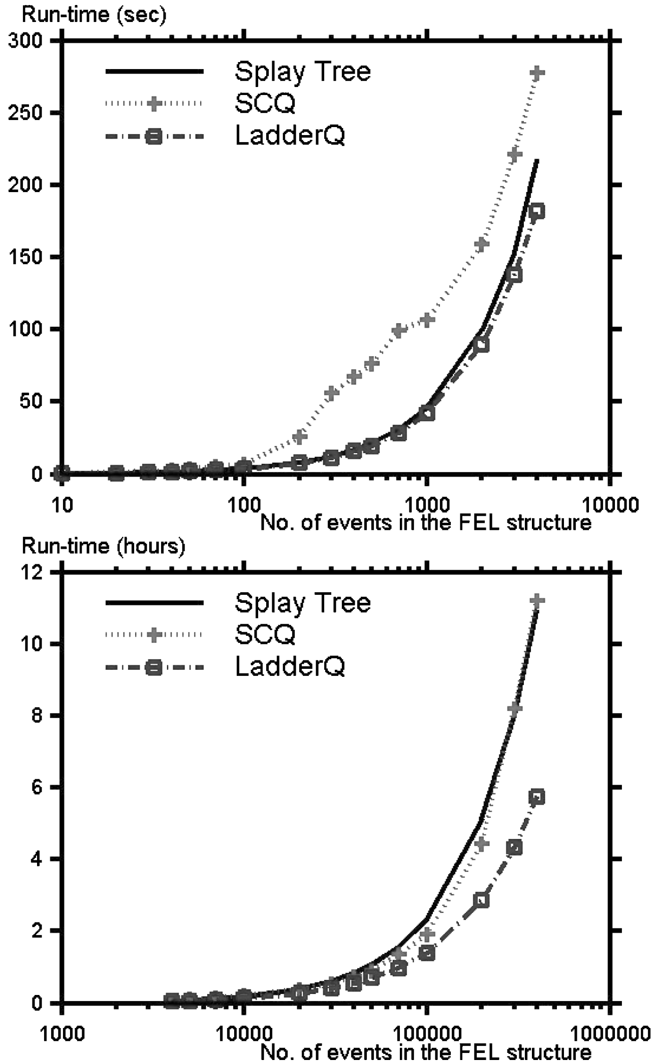


Fig. 9. Run-time performance measurements in SWAN for 10 to 400,000 network nodes.

SWAN simulation engine management and control time, in addition to the PES structure management time. So while the PES structure may be $O(1)$, the overall run-time is not expected to be $O(1)$. Nevertheless, the results illustrate a realistic view on the degree to which the performance of a PES structure can affect the run-time. The run-time for each simulation with different PES structure corresponds to taking the average value from five identical runs.

Figure 9 shows that for relatively small number of events, the cache memory of the hardware brings about good performance for both the simulations with the LadderQ and Splay Tree structure. As the number of events (or nodes) increases beyond 4,000, the LadderQ outperforms the Splay Tree and at 400,000, the speed increase is close to 100%. The SCQ on the other hand performs rather poorly. The reason for this poor performance is due to the initial onslaught of events with zero timestamps which sets off the SCQ size-based trigger frequently resulting in a long sublist in the SCQ's first bucket. However, by increasing the simulation time, it can be verified that the SCQ's performance gradually improves over that of the Splay Tree.

To this end, LadderQ has again shown to be a conspicuously superior PES structure particularly for large-scale simulation scenarios.

6. CONCLUSION

The choice of an efficient and stable pending event set implementation is of paramount importance in many large-scale simulations. Tree-based priority queues provide a stable performance regardless of event distribution but they are of $O(\log(n))$ complexity. On the other hand, the Sorted-discipline Calendar Queue (SCQ) is a multilist-based structure that provide $O(1)$ performance under some situations. However, the SCQ has been empirically shown to perform poorly under in scenarios where mean of the *jump* μ varies and/or when the number of events N fluctuates frequently by a factor of 2. This article proposes a new priority queue structure called Ladder Queue (or LadderQ) which not only exhibits the stability of tree-based structure but also the $O(1)$ characteristic that is often associated with a multilist-based structure. To achieve these properties, LadderQ does away with resize operations and uses a unique bucket-by-bucket copy operation via the rung spawning mechanism. Furthermore, in contrast to the SCQ, sorting is deferred as long as it is possible to reduce unnecessary operations. LadderQ's $O(1)$ average time complexity is also theoretically justified to be true in this article on the assumption that the mean of μ is finite and greater than zero, regardless of its variance. An extensive empirical study on the performance of LadderQ, in comparison with the Splay Tree and SCQ, is also presented. These empirical studies confirm the theoretical predictions that LadderQ is indeed $O(1)$. Furthermore we have shown empirically that LadderQ exhibits O(1) behavior even when the mean and variance of μ are *infinite*. LadderQ's stable and good $O(1)$ performance under all priority increment distributions for the Classic Hold and Up/Down experiments, for queue sizes ranging from 100 to 10 million events, demonstrates that it is the superior structure for small to large-scale discrete event simulation.

APPENDIX

The pseudo-code for LadderQ is given below.

```
void enqueue()
{
  if (TS >= TopStart) {
    insert into tail of Top
    NTop++;
    return;
  }
while (TS < RCur[x] && x <= NRung)
  x++;
```

```
  if (x <= NRung) {  /* found */
    bucket_k = (TS -- RStart[x]) / Bucketwidth[x];
    insert into tail of rung x, bucket_k
    NBucket[x, bucket_k]++;
    return;
  } else {
    if (NBot > THRES) {
      create_new_rung(NBot);
      transfer Bottom to it
      /* insert event in new rung */
      bucket_k = (TS -- RStart[NRung]) / Bucketwidth[NRung];
      insert into tail of rung NRung, bucket_k
      NBucket[NRung, bucket_k]++;
    } else {
      insert into Bottom using sequential search
      NBot++;
    }
  }
}
EVENT *dequeue()
{
  if Bottom not empty
    return next event from Bottom
  if (NRung > 0) {
    bucket_k = recurse_rung();
    if last bucket then NRung–;
    sort from bucket_k to Bottom
    return first event from Bottom
  } else {
    Bucketwidth[1] = (MaxTS -- MinTS) / NTop;
    TopStart = MaxTS;
    RStart[1] = RCur[1] = MinTS;
    transfer Top to rung 1 of Ladder
    bucket_k = recurse_rung();
    sort events from bucket_k and copy to Bottom
    return first event from Bottom
  }
}
BUCKET recurse_rung()
{
find_bucket:
  /* find next non-empty bucket from lowest rung */
  while (NBucket[NRung, k] == 0) {
    k++;
    RCur[NRung] += Bucketwidth[NRung];
  }
  if (NBucket[NRung, k] > THRES) {
```

```
        create_new_rung(NBucket[NRung, k]);
        recopy events from bucket to new rung
        goto find_bucket;
    }
    return k;
}
void create_new_rung(NEvent)
{
    NRung++;
    /* NEvent is the number of events to be recopied */
    Bucketwidth[NRung] = Bucketwidth[NRung-1] / NEvent;
    RStart[NRung] = RCur[NRung] = RCur[NRung-1];
}
```

REFERENCES

BROWN, R. 1988. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Commun. ACM 31*, 10 (Oct.), 1220–1227.

DAS, S., FUJIMOTO, R., PANESAR, K., ALLISON, D., AND HYBINETTE, M. 1994. GTW: A Time, warp system for shared memory multiprocessors. In *Proceedings of the 26th Conference on Winter Simulation* (Orlando, FL, Dec. 11–14). ACM, New York, pp. 1332–1339.

ERICKSON, K. B., LADNER, R. E., AND LAMARCA, A. 2000. Optimizing static calendar queues. *ACM Trans. Model. Comput. Simul. 10*, 3 (July) 179–214.

FALL, K. AND VARADHAN, K. 2002. *The NS Manual*. UCB/LBNL/VINT Network simulator v2. http://www.isi.edu/nsnam/ns/.

JONES, D. W. 1986. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM 29*, 4 (Apr.), 300–311.

MARIN, M. 1997. An empirical comparison of priority queue algorithms. Tech. Rep., Oxford University.

PARK, S. K. AND MILLER, K. W. 1988. Random number generators: Good ones are hard to find. *Commun. ACM 31*, 10 (Oct.), 1192–1201.

RÖNNGREN, R. AND AYANI, R. 1997. A comparative study of parallel and sequential priority queue algorithms. *ACM Trans. Model. Comput. Simul. 7*, 2 (Apr.) 157–209.

RÖNNGREN, R., RIBOE, J., AND AYANI, R. 1993. Lazy queue: New approach to implementing the pending event set. *Int. J. Comput. Simul. 3*, 303–332.

SCHWETMAN, H. 1996. CSIM18 User's Guide. Mesquite Software, Inc., Austin, TX.

SLEATOR, D. D. AND TARJAN, R. E. 1985. Self-adjusting binary search trees. *J. ACM 32*, 3 (July), 652–686.

TARJAN, R. E. 1985. Amortized computational complexity. *SIAM J. Alg. Disc. Meth. 6*, 2 (Apr.), 306–318.

Received February 2004; revised December 2004; accepted July 2005