MLIST: AN EFFICIENT PENDING EVENT SET STRUCTURE FOR DISCRETE EVENT SIMULATION

RICK SIOW MONG GOH IAN LI-JIN THNG

Department of Electrical and Computer Engineering
National University of Singapore
4 Engineering Drive 3, Singapore 117576

Abstract: In discrete event simulation (DES), the priority queue structure for managing the pending event set (PES) of the DES is known to account for as much as 40% of the simulation execution time. The priority queue contains events where the minimum time-stamp event has the highest priority and the maximum time-stamp event has the lowest priority. Current expected O(1) amortized time complexity priority queues are the Calendar Queue (CQ), Dynamic CQ, SNOOPy CQ, the 2-tier Dynamic Lazy CQ, as well as the 3-tier Lazy Queue. However, these priority queues require a costly resize operation that occurs when the number of events fluctuates or when the priority queue's operating parameters detect a skewed distribution and a resize operation is subsequently triggered. Each resize operation copies events from the old priority queue to a newly-created one. This article describes a novel and innovative approach using traditional linked lists with a multi-tier structure to develop an efficient priority queue that offers near O(1) performance by uniquely eliminating the resize operation. This new implementation, named Multi-tier Linked List (MList), is shown experimentally to be, on the average, at least 100% faster than all the current priority queues.

Keywords: pending event set, calendar queue, data structures, priority queue

1. INTRODUCTION

Discrete event simulation (DES) is a type of simulation that involves discrete models where a system is modeled as a number of concurrent logical processes interacting by a set of pending event messages. The DES utilizes a mathematical or logical model of a system which changes its state only at irregular epochs with finite number of precise points in time. Each point corresponds to an instance when at least one event has occurred. An event may change the state of the system and has an associated time-stamp that corresponds to the instance of its intended occurrence in the simulated time.

The pending event set (PES) is defined as the set of all events generated during a DES that have not been simulated or evaluated yet. The PES is essentially a priority queue controlling the flow of simulation of events with the minimum time-stamp having the highest priority and maximum time-stamp having the least priority. These events should be processed in non-decreasing time-order with multiple events of equal time-stamp being processed in the order that they are inserted into the PES. If processed in this manner, casuality relations between these events would not be violated, ensuring the simulation results to be correct and deterministic. Deterministic simulation results mean that exact duplicate results will be obtained when any two or more simulation runs with identical parameter settings are made.

It has been shown that up to 40% of the computational effort in a simulation may be devoted on the management of the PES alone [Comfort, 1984]. Henriksen [1983] concurs that in models of telecommunication systems, total run times easily differs by as much as 5:1, depending on the choice of the events list algorithm. Thus, as a demand for high-performance simulators grow, the importance of an efficient implementation of the PES becomes progressively vital.

A sequential DES frequently operates in a three-step cycle:

(1) Dequeue: Removal of an event with the highest priority from the PES.
(2) Execute: To process this dequeued event.
(3) Enqueue: Insertion of new event/s resulting from the execution into the PES.

Empirical study of actual DESs show that the dequeue and enqueue operations account for as much as 98% of all operations on the PES [Comfort 1984]. These two basic operations of maintaining the PES has run-time complexity closely dependent on the total number of events in the PES. Hence a PES should be efficient especially for large-scale simulations that involve large number of events during the execution of simulation models.

The most important metric of interest is the time required to perform the most common operations, which are the enqueue and dequeue operation, and this time is referred to as *access time* [Rönngren et al., 1997]. In DES, the common measure of interest is the *amortized* access time [Tarjan, 1985]. Tarjan defines amortized computational complexity as "to average the running times of operations in a sequence over the sequence." In other context such as real-time systems, the worst-case access is also of interest.

A common obstacle to an efficient implementation of the PES is the size of the PES. In general, larger PESs results in longer access time. Understandably, as the systems become larger, the length of simulation time may span from days to weeks. Simulations of modest sized telecommunication networks may take weeks on a modern workstation [Roberts, 1992]. Fine-grain simulations such as ATM (Asynchronous Transfer Mode) network simulations are time-consuming due to the huge number of events to process [Ahn et al., 1996]. Simulations of a 5-node high-speed ATM network took about 1,500 minutes for a 3-minute simulated time on a Sun4. Simulations of a wide-area gigabit network would take more than one week over a Sun Sparc 20 [Bahk et al., 1992]. It has also been reported that experiments of a 500-node network conducted in Tcpsim [Dupuy et al., 1990] for a 3-minute simulated time over Sun Ultra 1 took more than one day execution time on average [Oh et al., 1998].

An ideal implementation of the PES should be able to efficiently handle a widely varying PES size for small to large-scale DESs. By choosing a PES structure which is fast and efficient, it is possible to reduce the total execution time of simulation jobs significantly. To this end, many PES implementations have been proposed over the past decades, reminiscent of the selection of sorting algorithms available, in attempts to design the most efficient and time-saving priority queue for use in simulators. At the same time, there is still ongoing research in this area due to the reason that existing priority queue structures have certain shortcomings such as performing poorly under certain event distributions.

In this article, a simple yet novel approach of using multiple linked lists with a multi-tier structure is proposed. This new implementation, named Multi-tier Linked List (MList), is made up of a 3-tier structure in which the 1st tier T1 is a sorted single linked list, the 2nd tier T2 is a partially-sorted multi-list, and the 3rd tier T3 is an unsorted single linked list. The reason for using a multi-list structure is that tree-based structures are known to be bounded by O(log *n*) amortized time complexity whereas multi-list based structures have an expected O(1) performance under excellent operating conditions. Thus said, current multi-list based structures have not achieved such O(1) performance under practical situations. Therefore there is still room to achieve better performance as compared to a tree-based structure.

Through studying the various multi-list based structures (see Section 2), it is possible that the weaknesses faced by those structures be minimized or even eliminated. By incorporating deferred sorting methodology and eliminating the costly resize operations suffered by most O(1) priority queues, MList achieves impressive performance. Even though this structure bears semblance to the Lazy Queue, the mechanisms, complexities and performance of MList and Lazy Queue are entirely different. The rest of this article is organized as follows: Section 2, a brief discussion on current priority queue structures; Section 3, on MList's algorithm; Section 4, on the measurement methods and benchmarking used in the experiments; Section 5, the experimental results; finally in Section 6, the conclusion.

2. PRIORITY QUEUE DATA STRUCTURES

Several priority queue structures for the PES implementation are available in the literature. They can be characterized mainly into two sub-groups, tree-oriented and multi-list oriented priority queues. The Splay Tree [Sleator and Tarjan, 1985] was shown to be the most efficient tree-oriented implementation in [Jones, 1986; Rönngren and Ayani, 1997]. Multi-list oriented priority queue algorithms are mainly Two-Level [Franta and Maly, 1977], Ulrich's queue [Ulrich, 1978], Partitioned list structure [Davey and Vaucher, 1980], Calendar Queue [Brown, 1988], Lazy Queue [Rönngren et al., 1991, 1993], Dynamic Lazy Calendar Queue [Oh and Ahn, 1997], Dynamic Calendar Queue [Oh and Ahn, 1998] and SNOOPy Calendar Queue [Tan and Thng, 2000].

2.1. One-Tier Priority Queue

2.1.1. Splay Tree

The Splay Tree, developed by Sleator and Tarjan, is a heuristically balanced binary search tree. It uses a tree restructuring technique called splaying. Splaying is essentially a sequence of tree rotations that helps balance the tree by moving nodes on highly accessed branches upwards and closer to the root. The amortized time complexity of an enqueue operation in an *n*-node Splay Tree is guaranteed to be O(log *n*) while its dequeue operation is O(1).

2.1.2. Two-Level (TL)

The TL structure is an array of linked list with the last array serving as an overflow list. TL was shown

to perform worse than a Heap [Gonnet, 1976] for small events by McCormack and Sargent [1981]. However this is largely due to its large overhead as shown in McCormack and Sargent's experimental results. They suggested that it was mainly due to its large overhead and is not suitable for small scale simulations.

2.1.3. Calendar Queue (CQ)

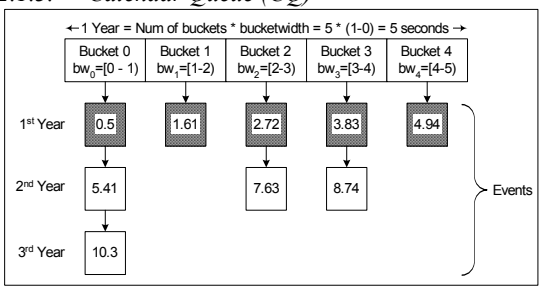


Figure 1: Standard Calendar Queue by Brown

According to Brown, Ulrich and Davey's implementation should be O(1) except for the presence of an overflow list. Their implementation of the overflow list caused the poor performance of the priority queues for skewed distributions, in which a significant percentage of events are enqueued in the overflow list. As such Brown's unique solution to this issue was to do away with an overflow list by employing the concept of a circular-year desk calendar; an array of linked lists without an overflow list (Figure 1). There are however problems associated with this O(1) priority queue.

Firstly, under highly-skewed distributions where most events fall into some few lists. CQ's ideal scenario is one in which there is 1 event in each of its array, which he called a "bucket", and that all events in CQ falls into the 1st year (i.e. shaded events in Figure 1). However in highly-skewed distributions, due to the poor sampling technique for a new suitable bucket-width after a resize, the events tend to fall only in a few buckets leaving most of the others empty.

Secondly, when the number of events in the PES fluctuates greatly. This occurs when there are many successive enqueues or dequeues that lead to frequent resizing of the number of buckets in CQ. During a resize, CQ would calculate its new bucket-width, re-initialize the multi-list to some number of buckets and the new bucket-width, and then recopies the old multi-list to the new one. This resize operation is a very costly overhead.

The CQ variants such as Dynamic CQ (DCQ) and SNOOPy CQ (SNOOPy) use different approaches of determining the optimum operating parameters of CQ to attempt to solve some of these problems. DCQ and SNOOPy are however experimentally found to have performed worse than CQ in some scenarios (see section 5).

2.2. Two-Tier Priority Queue

2.2.1. Dynamic Lazy CQ (DLCQ)

Oh and Ahn came up with DLCQ which is made up of an enhanced CQ as its primary tier and a secondary tier called Future EvenT list (FET). The enhanced CQ part is Brown's CQ with an additional Dynamic Resize (DR) algorithm to detect skewed distributions. Therefore in addition to CQ's static size-based resize, the DLCQ can also initiate a DR to dynamically change the bucket-width, even though the number of events remains the same. The FET structure is made up of two arrays called a root list and a leaf list. FET is partially-sorted and is used to store far future events during transient periods. Storing of far future events in FET helps to ease the additional frequent costly resizes due to the DR algorithm. However, DLCQ too was experimentally found to have performed worse than CQ in some scenarios, and performed better in others.

2.3. Three-Tier Priority Queue

2.3.1. Lazy Queue (LQ)

Rönngren et al. [1991, 1993] developed the 3-tier structure LQ that includes a near future (NF) linked list or binary heap (fully-sorted), a far future (FF) multi-list (partially-sorted) and a very far future (VFF) linked list (unsorted) that is used as an overflow bucket. The fundamental design of the LQ is to keep only a small portion of the events sorted in NF while letting far future events being unsorted in FF sub-lists and in VFF. There are 5 sets of criteria to determine if a resize should be triggered. After almost every dequeue or enqueue operation, some of the 5 sets of criteria have to be checked to determine if a resize should be made. The LQ will also have to ensure that after each resize operation, the rest of the other criteria are not violated. From a DES user point-of-view, there are 7 parameters that should to be set by the user before simulation begins. If not, the default static values would be used, at the expense of performance. Rönngren and Ayani [1997] had also made improvements to the LQ in which skew heaps were used as the NF and VFF in place of single linked lists. The resize operation was also modified to speed up small queue sizes.

Due to the highly complex algorithm and the source code being not obtainable from the authors, the LQ is not being considered in this article. However, the performance of LQ can be observed from the LQ literatures [Rönngren et al., 1991, 1993, 1997] and the conclusion drawn from these publications is that its performance is comparable to the CQ for most distributions. However the LQ is about twice better than the CQ for skewed distribution such as the Camel distribution. In highly skewed distributions

such as Change, which would be explained in later sections, the LQ performs much poorer than the CQ in some scenarios and in others, performs better. The results show that it is inconclusive whether the LQ is a superior priority queue as compared to the CQ.

3. THE MULTI-TIER LINKED LIST (MLIST)

The design of MList incorporates several improvements over CQ, DCQ, SNOOPy, DLCQ and LQ. Currently the performance of these priority queues is affected by the large overheads in the resize operation. During a resize operation, most or all the events are re-enqueued in a newly-created queue with new operating parameters, for example a new bucket-width. Note that the events in the CQ, DCQ and SNOOPy are already sorted before they are resized. To re-enqueue these sorted events may be highly inefficient. The DLCQ reduces the cost of the resize operation by reducing the number of events in the multi-list by storing the far future event in its FET. The LQ on the other hand utilizes the "lazy" methodology to sort events only on demand, leaving the far future events unsorted. Still the costly resize operations due to sampling for more accurate operating parameters and the frequent transfers of events back and forth between the tiers in both the DLCQ and LQ can be overwhelming.

A distinct feature of MList is that it does not resize. It works on the principle of a 3-tier structure to spread out events to handle skewed distributions. Sampling is also not required to obtain the appropriate bucket-width of its multi-list and this further reduces the overhead. MList also utilizes a deferred sorting methodology in which a small number of early time-stamp events are sorted, leaving the later events unsorted.

3.1. The MList Structure

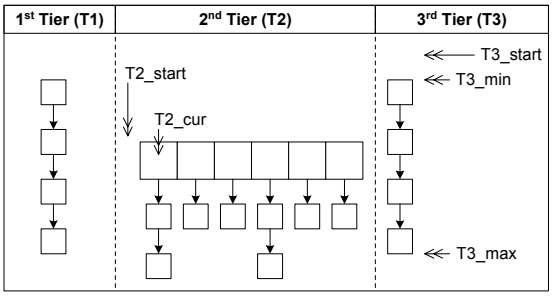


Figure 2: The MList Structure

The main building blocks of MList (Figure 2) are:

(1) 1st Tier (T1). T1 is a sorted linked list. This is the primary tier of MList in which the events with the minimum time-stamps are sorted and stored.

(2) 2nd Tier (T2). T2 is a partially-sorted multi-list structure. T2 stores events similar to the CQ's principle in which events are bucket-sorted. However, within each bucket which houses a linked list, the events are not sorted. And unlike the CQs which use a circular-year calendar concept in which the far future events are enqueued within the same bucket, MList employs a 3rd tier to contain those far future events. Hence T2 is strictly a one-year desk calendar.

3rd Tier (T3). T3 is an unsorted linked list. Acting as an overflow list to contain far future events, T3 buffers events that do not affect T1 and T2. This reduces the number of events in both T1 and T2. In addition, this last tier is the novelty behind the dynamic mechanism of MList.

3.2. Dynamic Mechanism of the Multi-list T2

In the CQs, the number of buckets is dependent on the number of events in their multi-list structure. If enqueues or dequeues give rise to the events exceeding or going below a certain threshold, their multi-list structure resizes to twice or half the number of buckets respectively. During each resize, the new bucket-width of those buckets is calculated using various sampling techniques or in the case of the SNOOPy, a moving average of the enqueue or dequeue cost. Besides adding complexity, this method of sampling adds to the overhead of a resize since a better sampling technique would mean that a larger number of events would be involved. For the case of the LQ, during a resize the number of buckets and bucket-width are halved or doubled according to the number of events in its T2 and T3.

MList marks the first departure from this heuristic method of determining the number of buckets as well as the bucket-width by making its multi-list T2 to be dependent on its T3. T2's length of 1 year, the number of buckets and the bucket-width are dynamically assigned only when T3 transfers events into T2. The MList keeps a set of variables to function and they are defined in Table 1. At the point of transfer of the events from T3 to T2, the length of 1 year in T2 is calculated as,

$$1\ Year\ in\ T2 = T3\_max - T3\_min \qquad (1)$$

and the bucket-width of each bucket in T2 will be,

$$T2\_bw = Bucket\text{-}width = \frac{1\,Year\ in\ T2}{T3\_num}, \quad (2)$$

where the number of buckets in T2 is equal to T3_num, giving an average of one event per bucket.

Note that if T1 and T2 are both empty of events, the occurrence of the first dequeue for this situation would again dynamically create a new T2 with the parameters calculated using equations (1) and (2). Table 1 lists a number of important variables to enable the reader to grasp a better understanding of MList.

Table 1: Definition of MList Variables

Name	Definition
T1_num	Number of events in T1.
T2_start	The minimum time-stamp of an event that can be enqueued in T2. This value is updated when events are being transferred from T3 to T2, with its value being set equal to T3_min at each T3 to T2 transfer.
T2_cur	The lower time-indexed of the earliest bucket in T2. E.g. (Figure 3), T2_cur is initially "0.1". Upon transferring of events from Bucket 0 to T1, T2_cur will be set to "1.1", and so on.
T2_num	Number of events in T2.
T2_bw	Bucket-width of T2.
T2_idx	The index of bucket array in which its events are to be transferred to T1.
T3_start	The minimum time-stamp of an event that can be enqueued in T3. This value will be set equal to T3_max at each transfer of events from T3 to T2.
T3_min	The minimum time-stamp in T3.
T3_max	The maximum time-stamp in T3.
T3_num	Number of events in T3.

3.3. The MList Algorithm

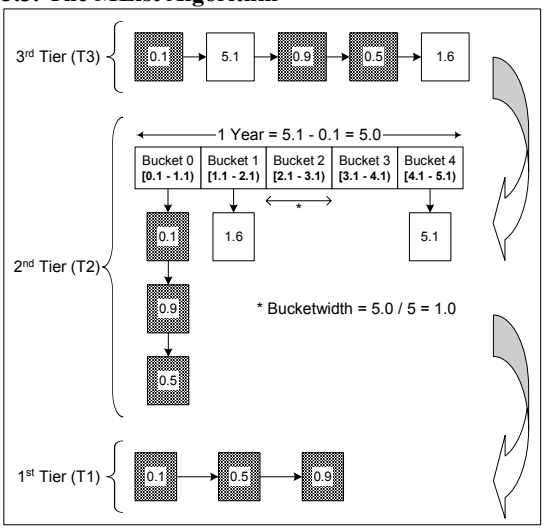


Figure 3: MList Queue Structure

3.3.1. Dequeue Operation

At the onset, all enqueued events are placed in T3 with no sorting required (Figure 3). On the first dequeue operation, all the events are transferred from T3 to T2 and the length of one year in T2, number of buckets and the bucket-width of T2 are dynamically assigned using Equations (1) and (2). Thereafter, events in Bucket 0 will be sorted in time-order using a standard sorting algorithm before being transferred to T1.

Figure 3 illustrates an example of how MList would function on its first dequeue or when there are no other events in T1 and T2. It depicts how the three smallest time-stamp events (shaded) are being transferred from T3 -> T2 and then T2 -> T1. Thereafter, the first dequeue would return "0.1" time-stamp event from T1. This is followed by "0.5" and then "0.9", assuming no other events are enqueued in T1.

Subsequently, events in Bucket 1 would be transferred to T1 and sorted. And since Buckets 2 and 3 are empty, the T2_idx would be incremented to Bucket 4. And so on until all the events in T2 are dequeued. After which, the whole cycle repeats itself with T3 treating the next dequeue alike the first dequeue as mentioned above. Again, the three parameters of T2 are dynamically assigned according to the events in T3. The pseudo-code of the dequeue operation is given in Figure 4.

```
struct event *dequeue() {
  Check_T1:
    if(T1_num > 0)
       Dequeue from T1; /*T1 is sorted*/
  //End of T1 check
  Check_T2: /*If T1 no event, check T2*/
    if(T2_num > 0) {
       T2_idx = 1st non-empty bucket; /*T2 has 1 year*/
       /*Increment T2_cur to the start of this bucket*/
       T2_cur=T2_start + T2_idx * T2_bw;
       Sort and transfer T2[T2_idx] events to T1;
        Dequeue from T1;
  //End of T2 check
  Check T3: /*iff T1 & T2 have no events*/
    if(T3_num > 0) {
       /*Calculate T2 bucket-width using Eqn(2) */
       T2_bw = (T3_max – T3_min) / T3_num;
       /*Update important variables*/
       T2_start=T3_min; T3_start=T3_max;
       T2_idx=0; T2_num=T3_num;
       Transfer all events from T3 into T2 buckets;
       goto Check_T2;
  //End of T3 check
} //Dequeue End
```

Figure 4: dequeue() Pseudo-code of MList

3.3.2. Enqueue Operation

For each enqueue operation, MList checks if that event time-stamp is greater than T3_start. If so, the event is simply placed at the end of the linked list in T3. If the event is not inserted in T3, MList then checks if the event time-stamp is greater than T2_cur. If so, the event is enqueued in T2. On enqueuing in T2, the index of the bucket (*bucket_idx*) where this event is to be inserted in T2 is;

$$Bucket\_idx = \frac{time\_stamp - T2\_start}{T2\_bw} \quad , \quad (3)$$

and the event is inserted at the back of the linked list corresponding to the bucket with index *bucket_idx*. Else, that event is enqueued in T1 via a linear insertion. The pseudo-code of the enqueue operation is given in Figure 5.

```
void enqueue(time_stamp, event) {
    if(time_stamp >= T3_start) {
       Insert at the back of T3;
       T3_num++; }
    else if(time_stamp >= T2_cur) { //Insert into T2
       /*Calculate bucket_idx using Eqn (3)*/
       bucket_idx = (time_stamp-T2_start) / T2_bw;
       /*Insert into T1[bucket_idx] as last element*/
       T2_num++; }
    else { //Insert into T1 via linear insertion
       Insert event in stable time-order
       T1_num++;
    }
} //Enqueue End
```

Figure 5: enqueue() Pseudo-code of MList

3.4. A Brief Time Complexity Analysis of MList

This section seeks to provide a brief performance analysis of MList using the amortized time complexity analysis [Tarjan, 1985].

Assume some events are enqueued into an empty MList. Initially all the events would be enqueued into T3. This enqueue of events into T3 is O(1) since T3 is an unsorted linked list and the events are simply inserted at the end of the linked list. On the first dequeue, all the events from T3 would be bucket-sorted into T2, where T2 is made up of buckets with an equal time-interval (bucket-width). Events would be distributed in T2 according to the event distribution. For instance, for a uniform distribution, each bucket would approximately hold one event. If there is only one event in a T2 bucket, the event would be dequeued immediately, bypassing T1. If however there are more than one event in T2, the events would be sorted and transferred to T1. Since the overhead of sorting and transferring of small number of events is small when amortized over all dequeues, it can be concluded that the dequeue operations in MList can be approximated to be O(1).

Suppose during these dequeue operations, more events are enqueued into MList, they would be expected to fall into the three tiers of MList with higher probability into T2 and T3. This is because the time-interval of T1 is equivalent to just one bucket in T2. Enqueuing of events into T2 or T3 gives O(1) complexity since T2 and T3 are made of unsorted linked lists and that the events are inserted at the end of the linked lists. However, enqueuing of events into T1 gives O(n_1) complexity since T1 is a sorted linked list and thus each event requires a sequential search for the correct position; n_1 is the number of events in T1. Table 2 summarizes the theoretical performance of MList.

Table 2: Amortized Time Complexity of MList Multi-tier Structure

	T1	T2	T3
Enqueue	O(n_1)	O(1)	O(1)
Dequeue	O(1)	-	-

Note: n_1 is the number of events in T1.

Specifically, under the uniform distribution, $n_1 << n$, where n is the total number of events enqueued into MList. This is because the time-interval of T1 is equal to the time-interval of only a bucket T2, i.e. time-interval of T1 << time-interval of T2, which translates to mean that the number of events in T1 << than those in T2. Therefore, regardless of the event distribution, if $n_1 << n$, then the amortized time complexity of enqueue operations in MList is expected to be O(1).

Under general event distributions, we can consider in terms of the probability of enqueuing an event in the various tiers in MList. The performance of MList is poor only in the scenario where many enqueues occur in T1 and where at the same time, n_1 is large. This poor performance is because T1 is maintained as a sorted linked list and each enqueue requires a linear search to place the event in the correct position. Let the probability of enqueuing an event in T1, bucket i in T2 and T3 be P_1, $P_{2,i}$ and P_3 respectively, where $0 \le i \le M$-1 and the number of buckets in T2 is M. $P_1 + \sum P_{2,i} + P_3 = 1$ and $P_1, P_{2,i}$ and $P_3 > 0$ for all i. MList has poor performance only if $P_1 \rightarrow 1$. However, for most distributions and simulation scenarios, P_1 does not tend to 1. Therefore, we can conclude that the performance of MList under general distributions is expected O(1). Through simulation benchmarks, we have shown empirically in Section 5 that MList has good performance for all the distributions tested.

4. PERFORMANCE MEASUREMENTS

In this section, we describe the design of the experiments which is essential in studying the performance of the priority queue structures under a wide variety of operating conditions.

4.1. Measurement Models

In order to find out the performance of the priority queues, the average access time for a queue under different loads has to be determined. The access time is referred to as the time taken for an enqueue

or dequeue operation. The parameters to be varied for each queue are: the access pattern, the queue size and the priority distribution. The access pattern models used emulate the steady-state and the transient phase. These two extreme cases are that which a PES would probably encounter in most DES. The two models to emulate the two scenarios respectively are as follow:

(1) Classic Hold model [Jones, 1986]. The queue to be benchmark is initially built up to the queue size to be tested by a random series of enqueues (higher probability) and dequeues. Thereafter a series of hold operations ensue. A hold operation is defined as a dequeue followed by an enqueue which value is the priority (time-stamp) of the event that was just dequeued and adding the priority increment distribution to be tested to this dequeued time-stamp. The average access time to be calculated is the average time taken for one hold operation. The number of hold operations in the benchmarks are set to be 1000 times the queue size tested. This method has advantages that:
   (a) The problem of determining the transient period is avoided, and
   (b) The transient period will affect to the same extent the different queue sizes tested [Rönngren and Ayani, 1997].
(2) Up/Down model [Rönngren et al., 1993]. An Up/Down is defined as the manner in which the queue is built up to a tested queue size by a sequence of enqueues. Thereafter followed an equally long sequence of dequeues. The number of Up/Down cycles is set to be 100. The average access time to be calculated is the time taken for all queue operations (enqueues and dequeues), divided by the total number of queue operations.

4.2. Priority Increment Distributions

The selections of typical distributions that occur in real simulation models are being employed. These distributions are commonly employed in [Rönngren et al., 1991, 1993; Oh and Ahn, 1997, 1998; Tan and Thng, 2000]. Four non-compound distributions, Rectangle (i.e. Uniform), Triangle, Negative Triangle and Camel(x,y) [Rönngren et al., 1991], and one compound distribution known as Change(A,B,x) [Rönngren et al., 1993]. The Camel(x,y) distribution is used to model bursty traffic in computer and communication networks which represents a highly-skewed distribution. The parameters used for Camel(x,y) results in two humps with x% of the probability mass being concentrated in the two humps. The duration of the humps makes up y% of an interval. Change(A,B,x) combines priority distribution A and B, with x priority increments being alternately drawn by A and B. Change(A,B,x) is to test a queue's the worst case behavior and its sensitivity to compound distributions. The four non-compound distributions are as shown in Figure 6.

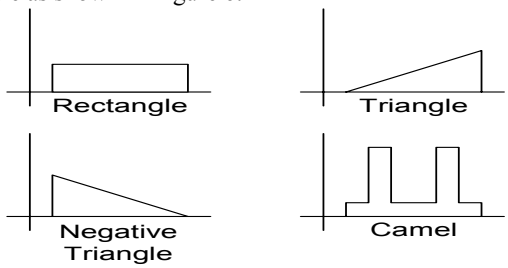


Figure 6: Non-compound Distributions Used

4.3. Benchmarking Test-bed

The experiments were carried out on an AMD Athlon MP 1.2GHz computer. This workstation has dual-processors but as the algorithms are sequential, did not made use of its true SMP capabilities. However this workstation ensured that when each benchmark was carried out, other background processes that might affect the results were kept at a minimum, thus obtaining more accurate experimental results.

10 runs of each experiment were done and the median value was obtained for each queue size simulated in the experiment. The median value is a measure of the central tendency and is chosen over the mean value because some background processes could have adversely affected a particular run of an experiment and averaging this value could render less accurate results.

Also, the experiments were performed with the required memory for each priority queue being pre-allocated. This was to eliminate the under-lying 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. A microsecond timer is used for all the experiments. Loop overhead time and the time taken for random numbers generated were removed using a dummy loop. All the code was written in the C programming language.

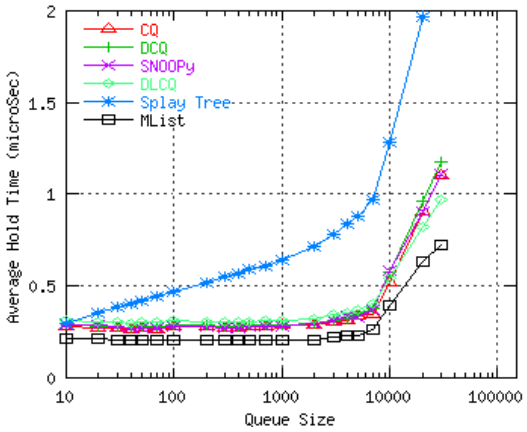
5. EXPERIMENTAL RESULTS

The objectives of this section are to present the performance of MList as compared with the current fastest multi-list priority queue data structures and to determine MList's generality and sensitivity in the five priority increment distributions using the Classic Hold and Up/Down models, as well as when the queue size increases from 10 to 30,000. Note

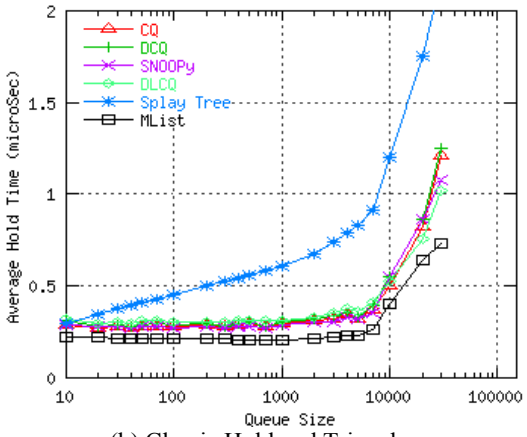
that a logarithmic scale has been used for the queue-size axis which leads to logarithmic complexity for linear plots.

5.1. Steady State Experiments

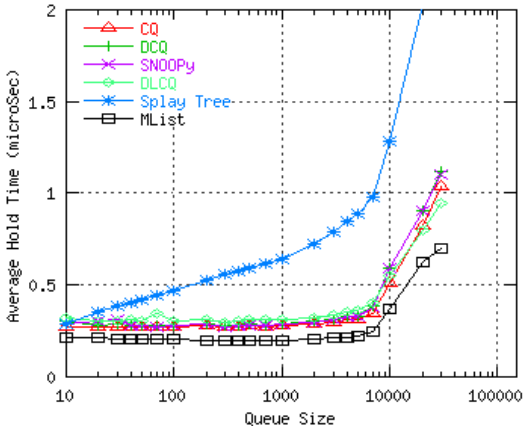
Figure 7 shows the results obtained under the Classic Hold model that is commonly employed to test the steady state performance of the priority queues. Table 3 summarizes with explicit figures of the performance gain (%), in terms of the distributions used in the experiments, of MList over the other priority queues.



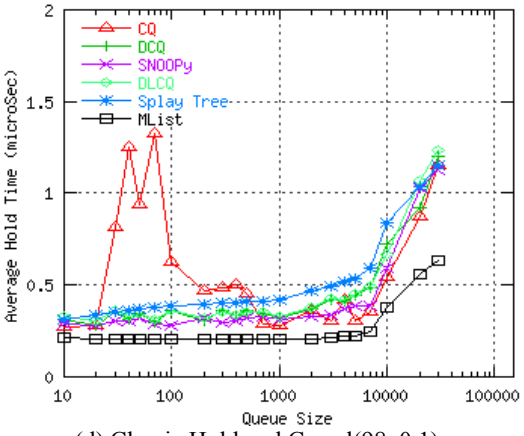
(a) Classic Hold and Rectangle



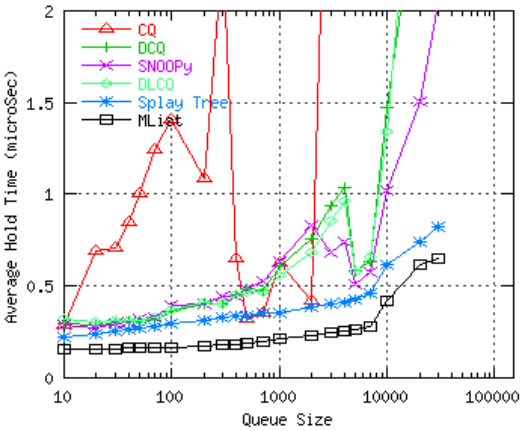
(b) Classic Hold and Triangle



(c) Classic Hold and Negative Triangle



(d) Classic Hold and Camel(98, 0.1)



(e) Classic Hold and Change(Camel(98,0.1),Triangle,2000)

Figure 7: Performance for Classic Hold Model Experiments

Figures 7(a) to 7(c) illustrate that the O(1) priority queues such as CQ and its variants perform with near O(1) performance and exemplify that the O(log *n*) Splay Tree pales in comparison with these distributions. MList shows better performance than the CQ and its variants due to its lower overhead in its management of the PES.

The obvious knee seen in the curves plotted is due to the declining cache performance and occurs when the queue size is about 10,000. This phenomenon is also observed in the graphs found in [Rönngren and Ayani, 1997] where the experiments were done on SUN and Intel architectures.

Figure 7(d) reveals some of the weaknesses found in the CQ. The CQ's performance is erratic due to its static size-based resize and that its sampling heuristics are unable to detect skewed distributions. The CQ variants have shown better performance due to their improved heuristics. The Splay Tree offers good performance, almost as well as MList.

Table 3: Average Performance Gain (%) of MList Over Other Priority Queues for Classic Hold Model

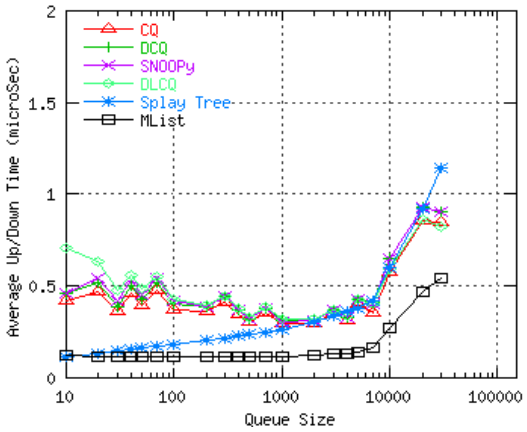
Distribution	CQ	DCQ	SNOOPy	DLCQ	Splay Tree
Rectangle	35.33	41.25	37.96	105.70	176.77
Triangle	36.06	41.78	34.27	102.72	157.77
Negative Triangle	36.67	44.01	43.92	111.30	187.57
Camel(98,0.1)	145.23	70.94	54.97	143.31	98.51
Change	1,264.67	169.08	140.83	292.78	62.87
Mean Value	***303.59***	***73.41***	***62.39***	***151.16***	***136.70***

Figure 7(e) which is based on a highly skewed distribution uncovers weaknesses found in all the CQ and its variants. Splay Tree, on the other hand, performs only slightly poorer at 62.87% worse than MList, as compared to over 100% for the CQ and its variants. DLCQ performs worse than the CQ variants due to the frequent transfers of events between its 2-tier structure, since a transfer is triggered according to the size of its primary tier.

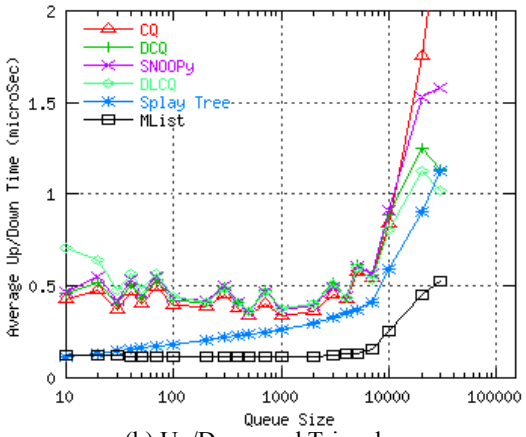
For the Classic Hold model, on the average, MList performs from 62% to 300% better than the other priority queues.

5.2. Transient State Experiments

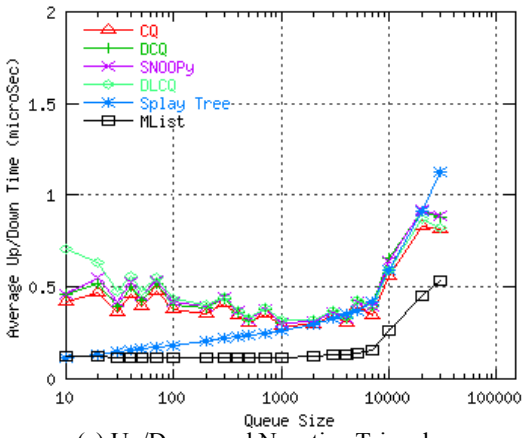
Figure 8 shows the results obtained under the Up/Down model that is generally used to test the transient state performance of priority queues. Table 4 summarizes with explicit figures of the performance gain (%), in terms of the distributions used in the experiments, of MList over the other priority queues.



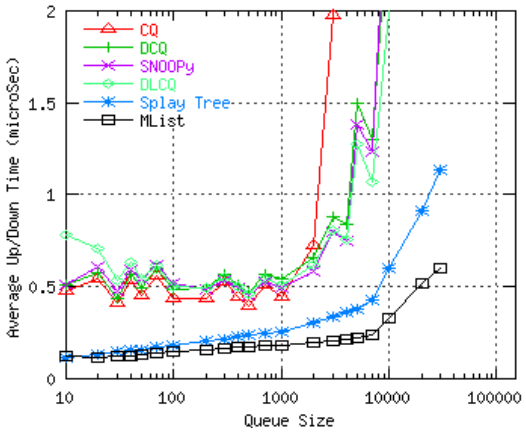
(a) Up/Down and Rectangle



(b) Up/Down and Triangle



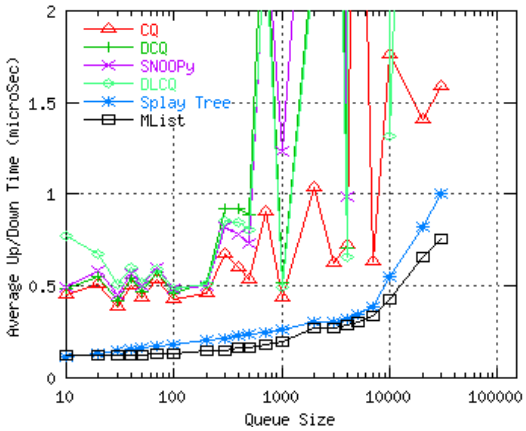
(c) Up/Down and Negative Triangle



(d) Up/Down and Camel(98, 0.1)

Table 4: Average Performance Gain (%) of MList Over Other Priority Queues for Up/Down Model

Distribution	CQ	DCQ	SNOOPy	DLCQ	Splay Tree
Rectangle	194.81	219.44	224.41	378.50	94.88
Triangle	261.14	271.67	283.51	441.34	96.47
Negative Triangle	190.90	215.93	221.28	374.08	94.45
Camel(98,0.1)	3,845.23	363.87	353.05	518.93	43.18
Change	284.33	1,671.89	7,681.86	1,427.47	25.24
Mean Value	***955.28***	***548.56***	***1,752.82***	***628.06***	***70.84***



(e) Up/Down and Change(Camel(98,0.1),Triangle, 2000)

Figure 8: Performance for Up/Down Model Experiments

The Up/Down model which tests the performance of priority queue structures during transient periods when the queue size fluctuates frequently, uncovers the weaknesses of the CQ and its variants. Figures 8(a) to 8(e) evidently shows that resize operations of the CQ and its variants are indeed costly. Whenever the queue size is just above (after an enqueue) or just below (after a dequeue) the powers of 2 queue sizes, i.e. 8 (2^3), 16 (2^4), …, 1024 (2^{11}), …, 16384 (2^{14}), and so on, the CQ and its variants would have to perform the resize operation. This is due to their static size-based algorithm being triggered.

The Splay Tree performs well and the plots clearly show its O(log *n*) is guaranteed in all queue sizes and distributions.

MList which does not have a resize operation, performs well even in highly skewed distribution as shown in Figures 8(d) and 8(e). The fact that the combination of its deferred methodology and the absent of resize operations by spreading out the events in the buckets, make it an efficient priority queue structure.

For the Up/Down model, on the average, Table 4 shows that MList performs from 70% to 1,700% better than the other priority queues.

Table 5: Average Performance Gain (%) of MList

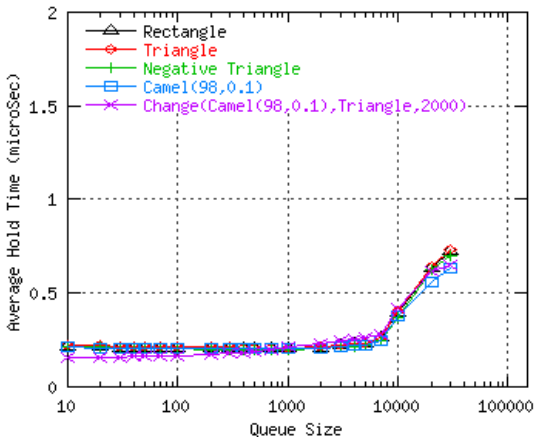
Priority Queue	Average Gain (%)
CQ	629.44
DCQ	310.99
SNOOPy	907.61
DLCQ	389.61
Splay	103.77

Table 5 shows the combined average for both the Classic Hold and Up/Down models to obtain an overall average performance gain (%), in terms of the queue size, priority increment distribution and the access pattern models, of MList over the other priority queue structures.

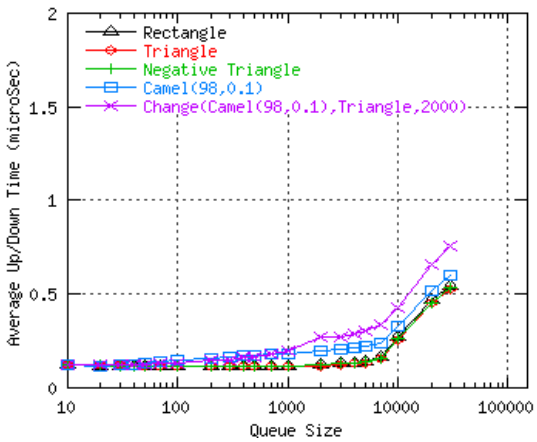
On the average, MList performs at least 100% better than the other priority queue structures.

5.3. Generality and Insensitivity of MList

Figures 9(a) and 9(b) shows MList generality and insensitivity under the various distributions and queue sizes. The figures show that MList is stable and exhibits near O(1) amortized time complexity for all scenarios. Under Change(Camel(98,0.1),Triangle,2000), MList shows signs of a slightly poorer performance because under a highly skewed distribution, the events in its 1st tier gets enqueued back into the 1st tier via a linear search for the correct position in the linked list. However, its low management overheads as well as the small number of events in its 1st tier due to its deferred sorting methodology ensures that MList performs more efficiently than the rest of the other priority queues under all distributions and all queue sizes.



(a) Classic Hold Model



(c) Up/Down Model

Figure 9: Generality and Insensitivity of MList

6. CONCLUSION

The current O(1) priority queues do not exhibit O(1) amortized time complexity in some of the scenarios tested under the commonly used models and distributions. A new priority queue implementation for the pending event set has been illustrated and tested with results showing performance surpassing current structures and consistently exhibits near O(1) performance in all distributions and queue sizes tested. MList's performance improvement of at least 100% better than all the current priority queues is owed to the fact it does not have the costly resize operations that severely affect the current priority queues. To handle skewed distributions, MList employs its multi-tier structure with deferred sorting methodology. These mechanisms contribute to the low management overheads and consequently ensure MList's superiority over the current O(1) priority queues, as well as the O(log *n*) Splay Tree. In addition, its simple algorithm, its generality to widely diverse queue sizes and its practically insensitivity to different distributions, adds to its potentiality in successfully and widely implementing this priority queue as the pending event set in discrete event simulations.

REFERENCES

Bahk, S. and Zarki M. E. 1992, "Dynamic multi-path routing and how it compares with other dynamic routing algorithms for high speed wide area network". In *Proceedings of ACM SIGCOMM '92 on Communications Architecture & Protocols*, 53-64.

Brown, R. 1988, "Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem". *Commun. ACM* 24, 12 (Dec.), 825-829.

Comfort, J. C. 1984, "The simulation of a master-slave event set processor". *Simulation* 42, 117-124.

Davey, D. and Vaucher, J. 1980, "Self-optimizing partition sequencing sets for discrete event simulation". *INFOR* 18: 41-61.

Dupuy, A., Schwartz, J., Yemini, Y. and Bacon, D. 1990, "NEST: a network simulation and prototyping testbed", *Commun. ACM 33*, 10 (Oct.), 63-74.

Franta, W.R. and Maly, K. 1977, "An efficient data structure for the simulation event set". *Commun. ACM* 20, 8: 596-602.

Gonnet, G.H. 1976, "Heaps applied to event driven mechanisms". *Commun. ACM* 19, 7: 417-418.

Jones, D. W. 1986, "An empirical comparison of priority-queue and event-set implementations". *Commun.ACM* 29: 300-311.

McCormack, W.M. and Sargent, R.G. 1981, "Analysis of future event-set algorithms for discrete event simulation". *Commun. ACM* 24, 12, 801-812.

Oh, S., and Ahn, J. 1997, "Dynamic Lazy Calendar Queue: An Event List for Network Simulation". In *Proceedings of HPC Asia '97*, 254-259.

Oh, S., and Ahn, J. 1998, "Dynamic Calendar Queue". In *Proceedings of the 32nd Annual Simulation Symposium*.

Roberts, J. W. 1992, "Performance evaluation and design of multiservice networks". Commission of the European Communities, Luxembourg.

Rönngren, R., Riboe, J. and Ayani, R. 1991, "Lazy queue: An efficient implementation of the pending event set". In *Proceeding of the 24th Annual Simulation Symposium*, 194-204.

Rönngren, R., Riboe, J., and Ayani, R. 1993, "Lazy Queue: New approach to implementing the pending event set". *Int. J. Computer Simulation* 3, 303-332.

Rönngren, R., and Ayani, R. 1997, "A Comparative Study of Parallel and Sequential Priority Queue Algorithms". *ACM Trans. Modeling and Computer Simulation* 7, 2(Apr) 157-209.

Sleator, D. and Tarjan, R. 1985, "Self adjusting binary search trees". *Journal of the ACM* 32: 652-686.

Tan, K.L., and Thng L-J., 2000, "SNOOPy Calendar Queue". In *Proceedings of the 2000 Winter Simulation Conference*, 487-495.

Tarjan, R.E. 1985, "Amortized computational complexity". *SIAM Journal on Algebraic and Discrete Methods*, 6(2), 306-318.

Ulrich, E.G. 1978, "Event manipulation for discrete simulations requiring large numbers of events". *Commun. ACM* 21, 9: 777-785.

BIOGRAPHY

RICK SIOW MONG GOH received his B.Eng. (Hons) from the Department of Electrical and Computer Engineering, National University of Singapore and is currently a research scholar pursuing his Ph.D. in the area of high performance sequential and parallel discrete event simulation. His research interests include distributed computing and object-oriented network simulators. He can be contacted at <engp1815@nus.edu.sg>

IAN LI-JIN THNG received his B.Eng. (with first class honours) from the University of Western Australia, Nedlands, in 1992 and the Ph.D. degree from the Curtin University of Technology, Bentley, Australia, in 1996. He is currently an Assistant Professor in the Department of Electrical and Computer Engineering at the National University of Singapore. Prior to that, he was a Postdoctoral Research Fellow in the Australian Telecommunications Research Institute, Australia. His research interests include very high-speed digital communications, array signal processing, provision of quality of service for handoffs in wireless cellular networks, optical burst switching and the development of high speed network simulators. He can be contacted at <eletlj@nus.edu.sg>