Proceedings of the 2000 Winter Simulation Conference
J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds.

SNOOPY CALENDAR QUEUE

Kah Leong Tan
Li-Jin Thng

Department of Electrical and Computer Engineering
10 Kent Ridge Crescent
National University of Singapore
Singapore 119260

ABSTRACT

Discrete event simulations often require a future event list structure to manage events according to their timestamp. The choice of an efficient data structure is vital to the performance of discrete event simulations as 40% of the time may be spent on its management. A Calendar Queue (CQ) or Dynamic Calendar Queue (DCQ) are two data structures that offers O(1) complexity regardless of the future event list size. CQ is known to perform poorly over skewed event distributions or when event distribution changes. DCQ improves on the CQ structure by detecting such scenarios in order to redistribute events. Both CQ and DCQ determine their operating parameters (bucket widths) by sampling events. However, sampling technique will fail if the samples do not accurately reflect the inter-event gap size. This paper presents a novel and alternative approach for determining the optimum operating parameter of a calendar queue based on performance statistics. Stress testing of the new calendar queue, henceforth referred to as the Statistically eNhanced with Optimum Operating Parameter Calendar Queue (SNOOPy CQ), with widely varying and severely skewed event arrival scenarios show that SNOOPy CQ offers a consistent O(1) performance and can execute up to 100 times faster than DCQ and CQ in certain scenarios.

1 INTRODUCTION

Discrete event simulations are widely used in many research areas to model a complex system's behavior. In discrete event simulation a system is modeled as a number of logical processes that interact among themselves by generating event messages with an execution timestamp associated with each of the messages. The pending event set (PES) is a set of all generated event messages that have not been serviced yet. A PES can be represented by a priority queue with messages with the smallest timestamp having the highest priority and vice versa. The choice of a data structure to represent the PES can affect the performance of a simulation greatly. If the number of events in the PES is huge as in the case of a fine-grain simulation, it has been shown that up to 40% of the simulation execution time may be spent on the management of the PES alone [Comfort, 1984].

A CQ is a data structure that offers O(1) time complexity regardless of the number of events in the PES. To achieve this, the CQ, which consists of an array of linked lists, tries to maintain a small number of events over each list. However, the CQ performs poorly when event distributions are highly skewed or when event distribution changes.

A DCQ [Oh and Ahn, 1999] has been proposed to solve the above-mentioned problem by adding a mechanism for detecting uneven distribution of events over its array of linked lists. Whenever this is detected, DCQ re-computes a new operating parameter for the calendar queue and redistributes events over a newly created array of linked lists.

Both the DCQ and CQ compute their operating parameter based on sampling a number of events in the PES. Sometimes the choices of samples are not sufficiently reflective of the optimum bucket width to use for the PES. When this occurs, performance of the DCQ and CQ degrade significantly and the newly resized calendar will not be able to maintain their O(1) processing complexity.

This paper proposes a novel approach in estimating an optimum operating parameter for a calendar queue. This approach is based on the past performance metrics of the calendar queue which can be obtained statistically. This approach provides an O(1) processing complexity for the calendar queue under all standard benchmarking distributions. It is also not susceptible to estimation error associated with the sampling method used in DCQ and CQ.

This paper is organized as follows. In section 2 we present in detail how a conventional CQ and DCQ operates, and their associated shortcomings. In section 3 we derive theoretically the optimum operating parameter

for a calendar queue. Utilizing the derived equations, section 4 describes the SNOOPy CQ mechanism. In section 5, the performance graphs of SNOOpy CQ, DCQ and CQ under different event arrival distributions are presented, compared and analyzed. Finally section 6 summarizes the contents of this paper and list down several recommendations for future work.

2 CQ AND DCQ

Sections 2 describes the operation of CQ and DCQ

2.1 Basic Calendar Queue Stucture.

Figure 1 illustrates the basic structure of a CQ consisting of an array of linked lists. An element in the array is often referred to as a bucket and each bucket stores several events using a single linked list structure. For notational conveniences, we define the following symbols:

N_B = Number of buckets in the CQ
B_W = Bucket width in seconds
D_Y = Duration of a year in seconds = $N_B \times B_W$
B_k = k^{th} bucket of the calendar queue where $0 \le k \le N_B - 1$

For example, in Figure 1, the CQ has $N_B=5$ buckets, i.e. B[0], B[1],…, B[4], each of width B_W = 1 second, representing an overall calendar year of duration D_Y = 5 seconds.

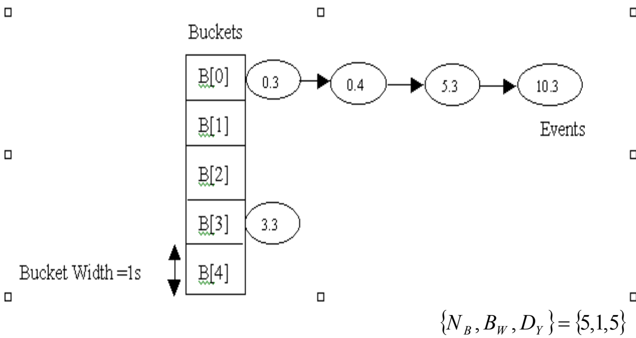


Figure 1: A Conventional Calendar Queue

To enqueue events with timestamp greater than or equal to a year's duration, a modulo-D_Y division is performed on the timestamp to determine the right bucket to insert the event. Therefore, any events falling on the same day, regardless of their year, is inserted into the same bucket and sorted in increasing time order as illustrated in Figure 1 and Table 1. To dequeue events, the CQ keeps track of the current calendar year and day it is in. It then searches for the earliest event that falls on the current year and day starting at bucket B[0]. If the event at the head node of the linked list at B[0] does not have the current year's timestamp, the search then turns to the head node of the linked list at B[1] and proceeds in this manner until B[N_B −1] is reached. When all the buckets have been cycled through, the current year will be incremented by 1 and the current day will be reset back to day 0 (i.e. bucket B[0]). For example, the event with timestamp 10.3 seconds in Figure 1 is only dequeued at the start of the third cycle.

Table 1: Event Timestamp Mapping

Event timestamp		Calendar Year	Calendar Day
0.3		0	1
0.4		0	1
5.3		1	1
10.3		2	1
3.3		0	4

2.2 CQ Resize Operation

To simplify the resize operation, the number of buckets in a CQ is often chosen to be of the power of two, i.e.

$$N_B = 2^n \text{ , } n \in Z, n \ge 0 \qquad (1)$$

The number of buckets are doubled or halved each time the number of events N_E exceeds $2N_B$ or decreases below $N_B/2$ respectively, i.e.

$$\text{If } N_E > 2N_B, N_B := 2 N_B$$
$$\text{If } N_E < N_B/2, N_B := N_B/2 \qquad (2)$$

When N_B is resized, a new operating parameter, i.e. B_W, has to be calculated as well. The new B_W that is adopted will be estimated by sampling the average inter-event time gap from the first few hundred events starting at the current bucket position. Thereafter, a new CQ is created and all the events in the old calendar will be recopied over. The resize heuristic obtained by sampling suffers from the following problems:

1) Since resizing is done only when the number of events doubles or halves that of N_B, this means that as long as N_E stays between $N_B/2$ and $2N_B$, the CQ will not adapt itself even if there is a drastic change in event arrivals causing heavily skewed event distributions to occur.
2) Sampling the first few hundred events starting at the current bucket position to estimate an appropriate bucket width is highly sub-optimal especially when event distributions are highly skewed.

2.3 DCQ Resize Operation

The DCQ improves on the conventional CQ by adding a mechanism to detect skewed event distributions and initiate a resize. The DCQ maintains two cost metrics C_E and C_E, where

C_E = Average Enqueue Cost
C_D = Average Dequeue Cost

The average enqueue cost is the average number of events that is required to be traversed before an insertion can be made on a linked list. The average dequeue cost is the average number of buckets that needs to be searched through before the event with the earliest timestamp can be found. The implementation aspects of updating the C_E metric and C_D metric is deferred until a later section. For the time being, it is sufficient to assume that these metrics are available. Now, a change in event distribution is detected whenever C_E or C_D exceeds some preset thresholds, e.g. 2, 3. If this should occur, DCQ initiates a resize on the width of buckets B_W, the number of buckets, N_B, remaining the same before and after the resize.

The DCQ structure also makes a small modification to the bucket width calculation of the CQ structure. Recall that for the case of CQ, the bucket width is estimated by sampling the first few hundred events of the current bucket. However, in DCQ, the bucket width is obtained by sampling the first few hundred events starting with the most populated bucket of the calendar queue structure. It is noted again that in the DCQ bucket width resize heuristic, sampling is again employed but this time on the most populated bucket. Therefore its performance is again dependent on how well the optimal inter-event gap size can be represented by these samples. If samples in the most populated bucket are constantly highly skewed, the DCQ resize operation is no better than the conventional CQ resize. This point is demonstrated later in our numerical studies presented in Section 6. In the next section, we will describe how SNOOPy CQ initiates a bucket width resize and then calculates the optimal bucket width.

3 SNOOPY CQ ALGORITHM

There are two parts to the SNOOPy CQ mechanism, namely, the ***SNOOPy triggering process*** which is responsible for initiating a bucket width resize and secondly, the ***SNOOPy bucket width optimisation*** process which is responsible for calculating the optimum bucket width when a resize operation has been initiated. As the ***triggering*** process is very much dependent on the ***bucket width optimisation*** process, we will proceed with explaining the second process first.

3.1 SNOOPy CQ Bucket Width Optimisation Process

The cost function that SNOOPy CQ aims to minimize when a bucket width resize is initiated is the sum of the average enqueue cost and average dequeue cost as follows:

$$\min_{B_W} C = C_E + C_{D,} \text{ subject to } N_B \text{ fixed} \quad (3)$$

The variable to optimize is the bucket width B_W. To optimize B_W, notice that if B_W is increased by a positive factor k, i.e. bucket width sizes are now larger in the system,

$$B_W := kB_W \quad (4)$$

then the average dequeue cost and the average enqueue cost are expected to increase and decrease respectively in the new queue. Hence the optimization problem in (3) transforms to the optimization of the factor k to minimize the following objective function:

$$\min_k C' = \min_k C_D' + C_E' = \min_k \frac{C_D}{g_1(k)} + g_2(k) C_E \quad (5)$$

where $g_1(k)$ and $g_2(k) \geq 1$ and have to be some monotonically increasing functions of k. In addition, $g_1(k)$ and $g_2(k)$ should also satisfy the following boundary conditions:

$$g_1(1) = g_2(1) = 1 \quad (6)$$

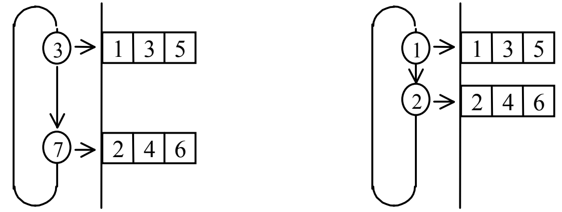
Note that the new average cost metrics C_D' and C_E' may remain optimized only for that short time period immediately after the bucket width upsize event has occurred, i.e queue distributions has not changed much before and after the upsize event. To handle a growing or declining PES scenario, more such optimizations can be triggered at appropriate times.

Now, the functions g_1 and g_2 not only depends on the event distribution of the queue at that particular instant, they may also depend on the factor k as well, i.e. different k factor upsize may demand different g_1 and g_2 functions. It is clear that to determine the exact functional in the face of statistical variations is not worthwhile. In order to proceed from this point forth, we take the approach of having no a priori knowledge of the event distribution and consider the best case and worst case cost decrements/increments after an upsize event. Once the bounds have been identified, an average objective function can be established for optimizing k.

For the case of the average dequeue cost, we note that increasing the bucket width packs events together. Hence the new average dequeue cost $C_D^{'}$ (within that short time period after the upsize) should range between

$$\frac{C_D}{k} \leq C_D^{'} \leq C_D \tag{7}$$

The upper bound in (7) indicates that in the worst case, there may be no reduction to the average dequeue cost even if the bucket width is increased. Such a scenario may occur as illustrated in Figure 2 where events are concentrated in only two buckets, i.e. 3 and 7, and events have time stamps such that the dequeue mechanism must alternate between these two buckets for every event that is dequeued. In Figure 2, increasing the bucket width moves the two bucket of events together but leaves a longer tail of empty buckets in the new calendar queue. As the old queue and the new queue have the same number of buckets N_B, it is clear that the number of empty buckets that is traversed so as to dequeue alternate events (residing respectively in the two buckets) is exactly the same. Conversely, the lower bound in (7) indicates the most ideal average dequeue cost reduction when the bucket width is upsized by *k*, subject to this condition - ***that the upsize does not cause the onset of a degenerate queue structure***. A degenerate queue structure occurs when *k* is so large such that after resizing, all the elements are merged into a single bucket. Consequently, the average dequeue cost decreases to 0 but the calendar queue degenerates into a single linked list structure which is undesirable. To avoid the degenerate scenario, the lower bound for the reduction in the average dequeue cost has to be constrained (which will in turn limit the size of *k*). Now, the best possible reduction only occurs, without the onset of degeneration, when the *k* factor upsize causes the distance between the previous linked list structures to be *k*-times closer to each other in the new queue structure but does not cause any of the previous linked list structure to merge, and all events dequeued belong to the current year so that there is no need to traverse the tail of empty buckets. Under this ideal scenario, we note that upsizing the bucket width by *k* would cause the number of empty buckets between filled buckets to be divided by *k*. Hence each subsequent dequeue operation in the new structure would traverse *k*-times less empty buckets compared to previous traversals in the old queue.



Before Upsizing Bucket After Upsizing Bucket

Figure 2: Worst case C_D reduction after bucket width upsizing

Increasing the bucket width merges events, resulting in longer linked lists in the new calendar queue structure. Hence the new average enqueue cost $C_E^{'}$ (within that short time period after the upsize) should increase and range between

$$C_E \leq C_E^{'} \leq kC_E \tag{8}$$

The lower bound in (8) indicates the best case situation in that the enqueue cost does not increase after the upsizing. Such situations occur when the upsize factor is not large enough to cause linked list structures of the previous queue to merge. Consequently, the linked list structures of the old queue are all preserved in the new queue. The only difference is that the new linked list structures are now assigned to buckets with smaller indexes (which affects the dequeue cost but not the enqueue cost). Conversely, the upper bound in (8) indicates that in the worst case situation, the average enqueue cost increases *k* times its previous. This situation occurs when prior to the upsizing, all non- empty buckets are clustered to each other as shown in Figure 3. After the upsizing, all the events should now be found in a cluster of buckets which is *k*-times smaller. Since N_E is identical in that short time before and after the bucket upsize, the length of each linked list in the new queue should on average grow by k.

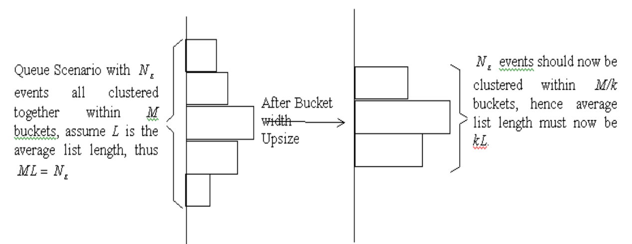


Figure 3: Worst case C_E increase after bucket width upsizing

With the bounds for $C_D^{'}$ and $C_E^{'}$ defined in (7) and (8), these bounds can be permutated to form four possible limiting cases of cost decrements/increments after a bucket upsize event. Taking the average of these four possible

permutations, we obtain the following average objective function for optimizing k.

$$C_D^{'} + C_E^{'} = \frac{1}{2}\left(\frac{C_D}{k} + C_D\right) + \frac{1}{2}\left(kC_E + C_E\right) \quad (9)$$

Notice that the cost function in (9) satisfies the boundary conditions in (6). Differentiating (9) with respect to k to solve for the minimum cost, we obtain the following optimal relations:

$$k = \sqrt{\frac{C_D}{C_E}},\ C_D^{'} = \frac{\sqrt{C_D C_E}}{2} + \frac{C_D}{2},\ C_E^{'} = \frac{\sqrt{C_D C_E}}{2} + \frac{C_E}{2} \quad (10)$$

Hence, the optimal bucket width to use for upsizing the bucket width is

$$B_W^{*} = \sqrt{\frac{C_D}{C_E}} B_W \quad (11)$$

It can be easily verified that for the case of downsizing the bucket width, an identical average cost function to (9) can be derived where k is now less than or equal to unity. Consequently, the same set of optimal solutions shown in (10) also applies for a bucket width downsizing event.

3.2 SNOOPy CQ Bucket Width Resize Triggering Process

As the SNOOPy CQ triggering process depends on C_E and C_D , a short explanation on how C_E and C_D is practically obtained is presented. The SNOOPy CQ initiation process keeps track of two types of average cost. The first is a ***slot average*** and the second is a ***multi-slot moving average***. The following definitions explain:

- Slot : a time interval corresponding to N_B dequeue operations or N_B enqueue operations and not any mixture of both.
- $C_{D,1}$: average dequeue cost averaged over 1 slot of dequeue operations. Memory effects associated with $C_{D,1}$ from slot to slot is zero, i.e. each slot derives a new $C_{D,1}$ based only on dequeue operations occurring during the current slot period.
- $C_{E,1}$: average enqueue cost averaged over 1 slot of enqueue operations. It has similar properties as $C_{D,1}$.
- Era : a time interval between two consecutive bucket resize events.
- $C_{D,n}$: a moving average of n consecutive $C_{D,1}$'s obtained in an era. When an era begins, the first n consecutive $C_{D,1}$'s are averaged to obtain $C_{D,n}$. Thereafter, any new $C_{D,1}$ that is generated would be included into the moving average after the oldest $C_{D,1}$ has been discarded. There is no memory effect associated with $C_{D,n}$ from era to era. If the era is less than n slots, then $C_{D,n}$ is zero throughout that era.
- $C_{E,n}$: a moving average of n consecutive $C_{E,1}$'s obtained in an era. It has similar properties as $C_{D,n}$

In the case of DCQ, only $C_{D,1}$ and $C_{E,1}$ are tracked while the SNOOPy CQ structure tracks $C_{D,1}$, $C_{E,1}$, $C_{D,10}$ and $C_{E,10}$.

The SNOOPy CQ adopts all the triggering mechanisms of the conventional CQ and DCQ structure and adds another two more triggering mechanisms, namely

$$C_{E,10} \geq 2 \times C_{D,10} \text{ or } C_{D,10} \geq 2 \times C_{E,10} \quad (12)$$

This means that when the 10-slots moving average cost factors differ by a factor of 2, a bucket resize is also initiated by SNOOPy CQ. The use of a 10-slots moving average has been found in our simulations to provide enough stability in the average costs to strike a good balance between excessive triggering and un-responsive triggering. The use of the triggering condition in (12) results from the optimal cost solutions shown in (10) where it is noted that if the current average costs C_D and C_E already satisfies the optimal conditions, i.e.

$$C_D = \frac{\sqrt{C_D C_E}}{2} + \frac{C_D}{2} \text{ and } C_E = \frac{\sqrt{C_D C_E}}{2} + \frac{C_E}{2} \quad (13)$$

then there is no necessity for a bucket resizing event. Solving the equations simultaneously in (13), we obtain the unique and more simplified condition that if

$$C_D = C_E \quad (14)$$

then there is no need for a bucket width resize event. Hence the objective of the triggering mechanism in (12) is to equalize C_D and C_E within some tolerance factor (i.e. 2).

It is noted that adding two more triggering mechanisms for the SNOOPy CQ structure in (12) does not necessarily imply that the SNOOPy CQ will resize itself more often than the DCQ structure. In fact, our simulations show that the SNOOPy CQ resizes less often than the DCQ structure and the main reason is that the SNOOPy CQ uses a more superior bucket width optimization calculation than DCQ's sampling technique, consequently, the SNOOPy CQ operates most of the time in its optimum state keeping both the DCQ-inherited and SNOOPy CQ triggering mechanisms inactive.

4 FINE-TUNED SNOOPY CQ ALGORITHM

The SNOOPy CQ algorithm should be employed judiciously especially when a new calendar queue era has just started after a complete resize. This is because any performance metrics corresponding to the new era will not be sufficiently reflective of the queue performance unless there is sufficient amount of dequeue operations D_{ops} and likewise, sufficient amount of enqueue operations E_{ops}. Note that D_{ops} affects C_D and likewise, E_{ops} affect C_E. Hence some fine tuning is required and this is reflected in the pseudo-codes of the SNOOPy CQ Enqueue() function as illustrated in Figure 4. Line 12 of the pseudo- codes show how it is decided whether to use the SNOOPy CQ bucket width calculation or the DCQ bucket width technique (which is based on sampling around the most populated bucket). The Calendar_Resize(B_W, N_B) function, which is referenced in the Enqueue() function, copies events in the old calendar queue to a new calendar queue consisting of N_B buckets, each with width B_W. The Calendar_Resize() function also incorporates a Resize(uneven) module which may further fine tune the new queue structure. The usefulness of the Resize(uneven) module to further fine tune a newly created calendar queue is mentioned in the DCQ literature [Oh and Ahn, 1999]. Note that the resize triggers are found only in the Enqueue() and Dequeue() functions as these functions manage the events of the queue. The differences between the Dequeue() function and the Enqueue() function is illustrated in Figure 5.

```
Enqueue(){
(1)Enqueue new event to the appropriate bucket
and update AccEvSkip ; /* AccEvSkip accumulates
the number of events skipped for each enqueue
operation since the enqueue slot began. For the
case of the Dequeue() function, another
variable, AccBuckSkip, is used to accumulate
the number of empty buckets traversed for each
dequeue operation since the dequeue slot began
*/
(2)N_E++;
(3)if(N_E>2N_B){// CQ trigger for a growing PES
(4)   N_B=2N_B;
(5)   B_W:=Use Sampling Method;
(6)   Calendar_Resize(B_W,N_B);/* After a resize, a
      new era begins, therefore, we set … */
(7)    C_D,1=C_E,1=C_D,10=C_E,10=E_ops=D_ops=0;
AccEvSkip=AccBuckSkip=0;}
   else{
(8)  E_ops++;
/*Track the number of enqueue operations since
the slot started*/
(9)  if(E_ops>N_B){//end of an enqueue slot
(10)    Update C_E,1 and C_E,10; //Update costs
(11)    if(C_E,1>2 or C_D,10> 2C_E,10 or C_E,10> 2C_D,10){
         /* After trigger check which
         bucket width algorithm to use */
(12)       if(E_ops>64 && D_ops>64){/*enough samples
              use Snoopy CQ*/
(13)          if(C_E,1>2) //DCQ inherited trigger

(14)             C_E= C_E,1, C_D=AccBuckSkip/D_ops;
/* C_D,1 may not be available at this time */
              else//This is a Snoopy CQ trigger
(15)             C_E= C_E,10, C_D= C_D,10;
/* Now obtain the new SNOOPy CQ bucket width */
(16)             B_W := B_W * sqrt(C_D / C_E) ;}

            else//not enough operations, use DCQ
(17)           B_W:=Use Sampling Method;
(18)         Calendar_Resize(B_W,N_B);
/* A calendar resize marks the end of an era,
so we set …*/
(19)         C_D,1=C_E,10=C_D,10=D_ops=0;
             AccBuckSkip=0;
        }
/*end of pseudo-codes dealing with a trigger
condition*/
(20)    C_E,1=E_ops=AccEvSkip=0;
/* Since this is also the end of a slot of en
queue operations*/
     }// end of pseudo-codes for end of slot

   }
(21)Return;}
```

Line (16): $B_W := B_W \sqrt{C_D / C_E}$

Figure 4: Enqueue() Pseudo Codes of SNOOPy CQ

5 EXPERIMENTS AND RESULTS ANALYSIS

The classical Hold and Up/Down model are used to benchmark the performance for a conventional calendar queue (SCQ), DCQ and SNOOPy CQ. The priority increment distributions used are the Rect, Triag, NegTriag, Camel(x,y) and Change(A,B,x) distributions as were used by Oh and Ahn [1999] and Rönngren et al.[1993]. Camel(x,y) represents a 2 hump distribution will x% of its mass concentrated in the two humps and the duration of the two humps is y% of the total interval. Change(A,B,x) interleaves two priority distribution A and B together. Initially x priority increments are drawn from A followed by another x priority increments drawn from B and so on. The shapes of the priority increment distributions used are shown in Figure 6.

Line	Dequeue() replaces it with ….
1	Dequeue event from the head of the appropriate bucket and update AccBuckSkip;
2	$N_E --$;
3	if ($N_B > 2N_E$) { /* CQ trigger for a declining PES */
4	$N_B := N_B / 2$;
8	D_{ops} ++; /* Track the number of dequeue operations since the slot started */
9	if ($D_{ops} > N_B$) { // end of a slot, update costs, check triggers, resize if necessary.
10	Update $C_{D,1}$ and $C_{D,10}$;
11	if($C_{D,1}$ >2 or $C_{D,10}$ >2 $C_{E,10}$ or $C_{E,10}$ >2 $C_{D,10}$) {
13	if ($C_{D,1}$ > 2) // This is a DCQ-inherited trigger
14.	$C_D = C_{D,1}$, C_E = AccEvSkip / E_{ops} ; /* $C_{E,1}$ may not be available at this time */
19	$C_{E,1} = C_{E,10} = C_{D,10} = E_{ops}$ = AccEvSkip = 0 ;
20	$C_{D,1} = D_{ops}$ = AccBuckSkip = 0 /*Since this is also the end of a slot of dequeue operations*/

Figure 5: Differences between Dequeue() and Enqueue() Pseudo-codes

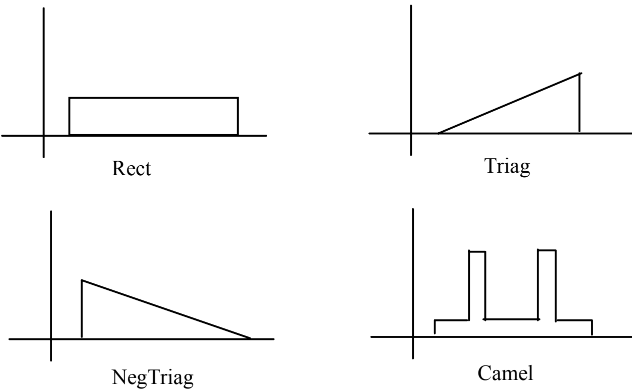
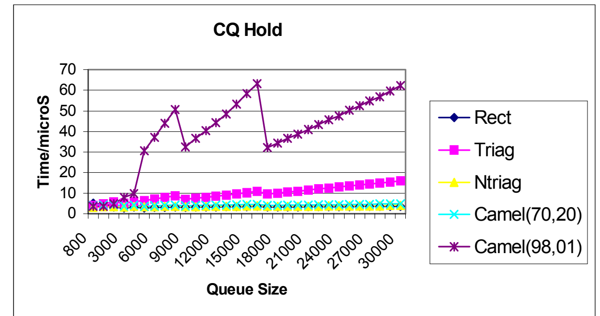
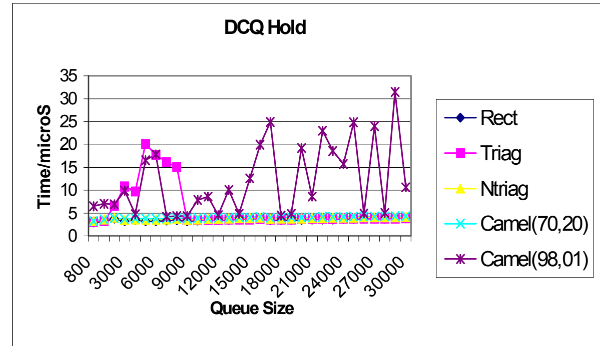


Figure 6: Benchmarking Distributions

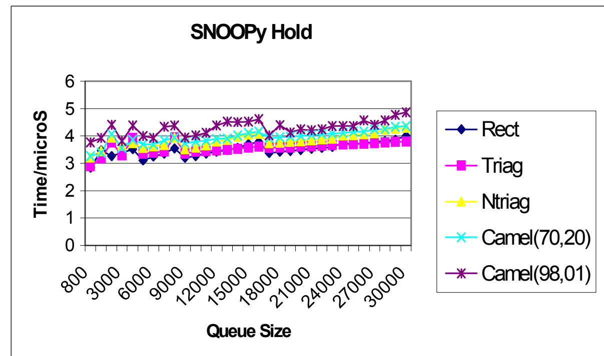
The Classical Hold and Up/Down model represent two extreme cases and are frequently used to show the performance bounds of PES implementations [Vaucher and Duval, 1975]. The number of hold operations performed is 100 × the queue size. Loop overhead time is eliminated using another dummy loop as was described by Rönngren and Ayani[1997]. The experiment is done on an AMD K6 210Mhz (83×2.5) with 32Mb RAM system running Windows 95. Figure 7 shows the Hold results under different distribution for CQ, DCQ and SNOOPy CQ.



(a)



(b)



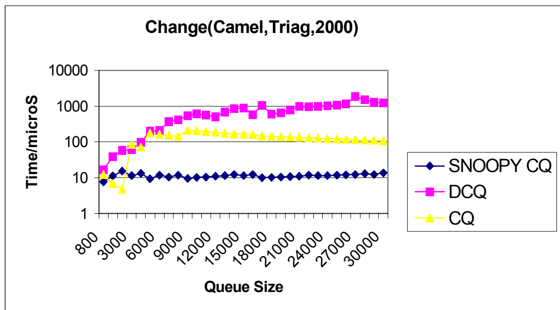
(c)

Figure 7: Average time per Hold operation for CQ, DCQ and SNOOPy CQ

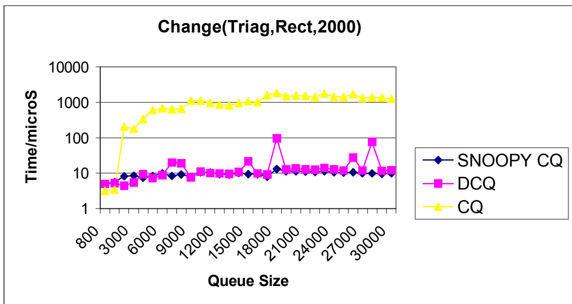
It can be observed that out of the three queue implementations, SNOOPy CQ is the least affected by the type of distribution used. It boasts average hold times between 3 to 5 µs for all priority increment distributions. The DCQ performance is erratic especially for the Triag and Camel(98,01) distributions. Average hold times vary from 3 to 30 µs. The CQ performance is the worst among the three queue implementations with average access times varying from 3 to 65 µs. It is most affected by the Triag and Camel(98,01) distributions. Both DCQ and CQ suffer from the same problem of estimating the optimum bucket width just by event sampling. For DCQ, event sampling around the most populated bucket seems to give a good

estimate for some situation but not every situation. Thus, the inconsistent performance as shown in Figure 7(b).

Two other distributions used for the Hold benchmarking test are the Change(camel9801(9-10),Triag(0- 0.0001),2000) and the Change(Triag(9-10),Rect(0- 0.0001),2000). Camel9801(9-10) represents the camel(98,01) in the range of 9 to 10. Triag(0-0.0001) distribution represents the Triag distribution in the range of 0 to 0.0001. Triag(9-10) represents the Triag distribution in the range of 9 to 10, and finally the Rect(0-0.0001) represents a Rect distribution in the range of 0 to 0.0001. The results of the Hold benchmarks are shown in Figure 8.



(a)

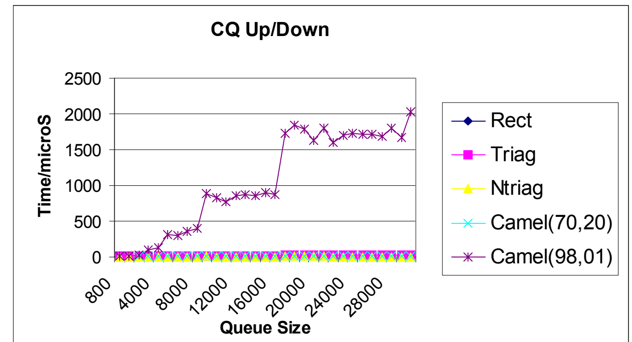


(b)

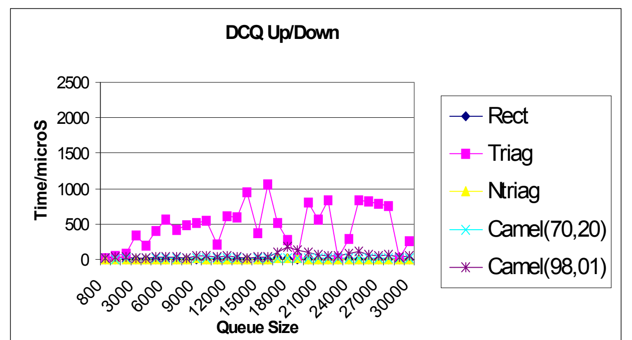
Figure 8: Average time per Hold operation under Change(A,B,x)

From these two graphs it can be seen that SNOOPy CQ adapts to changes in distribution easily with average hold time in the range of 10µs for Figure 8(a) and 8(b). The resize heuristics for CQ and DCQ fail miserably for (a), with average hold time of 100µs and up to 1000µs. In (b), the DCQ heuristic could adapt itself for certain queue sizes but not all. Average hold time ranges from 10µs to 100µs. CQ, on the other hand, fails to adapt at all due to its static resize algorithm. Average hold time deteriorates to 1000µs for large queue sizes. Again from these two graphs, it is evident that estimating an optimum bucket width to use just by event sampling does not guarantee consistent performance under all situations. This is unlike the more superior SNOOPy CQ resize heuristic.

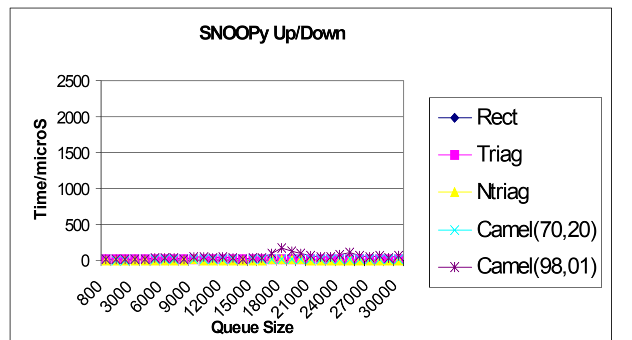
For the Up/Down model, a total of 10 cycles of filling up the calendar to reach the required queue size followed by a complete emptying of the calendar was done. The average time per enqueue/dequeue operation is then computed and plotted against different queue sizes. The plots for CQ, DCQ and SNOOPy CQ under different priority increment distributions are given in Figure 9.



(a)



(b)



(c)

Figure 9: Average time per enqueue/dequeue operation under Up/Down Model

Figure 9(a) shows that the CQ resize heuristic is sensitive under Camel(98,01) distributions despite many resize operations. This is because the CQ structure is unable to determine the optimum bucket width by event sampling.

Figure 9(b) shows that the DCQ resize heuristic works well under most distributions except Triag. This is because the heuristic tend to estimate a bucket width that is too

small since it samples events around the most populated bucket.

Figure 9(c) shows that the SNOOPy CQ performs well under all distributions and is not susceptible to underestimating or overestimating the optimum bucket width to use.

Finally, Figure 10 illustrates the effectiveness of the DCQ resize heuristic compared to the SNOOPy CQ heuristics in terms of the number of resize triggers. Recall earlier that the SNOOPy CQ algorithm adds two more triggering mechanism and it was mentioned that it does not necessarily mean that SNOOPy CQ initiates a resize more often. The plots in Figure 10 shows that on average, SNOOPy CQ takes 50% less resize operations to achieve optimal operating parameters compared to DCQ for the case of the Camel(98,01) distribution in the Hold scenario. Other distributions used for the Hold scenario are well behaved and do not cause DCQ and SNOOPy CQ to trigger often enough to provide meaningful comparisons on the number of resize operations.

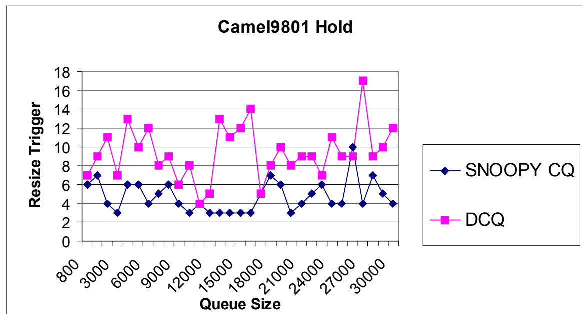


Figure 10: Number of Resize Triggers in the Camel(98,01) Hold scenario

6 CONCLUSION

Choosing the correct PES data structure for a simulator is important for speeding up huge sized simulations. Calendar Queue and Dynamic Calendar Queue are two data structure that are often used to implement the PES. Both of these data structures perform well under some situation but badly in others. The resize heuristic of CQ and DCQ could not guarantee a good estimate of an optimum bucket width to use under all situations. This paper proposes a novel approach in estimating the optimum bucket width to use based on performance statistics of the calendar. The data structure employing this approach is called Statistically eNhanced with Optimum Operating Parameter Calendar Queue (SNOOPy CQ). It has been demonstrated that this technique provides a superior bucket width estimate to use during a resize event. Experimental results from the Hold and Up/Down model show that SNOOPy CQ consistently offers O(1) time complexity under different distributions, unlike CQ and DCQ. In certain scenarios, SNOOPy CQ has been shown to be 100x faster than CQ or DCQ. In more well-behaved queue distributions, the SNOOPy CQ has the same order of performance compared to CQ and DCQ.

REFERENCES

Comfort, J.C., 1984. The simulation of a master-slave event set processor. *Simulation 42, 3 (March), 117-124.*

Oh, S., and Ahn, J.. 1999. Dynamic Calendar Queue. In *Proceeding of the 32nd Annual Simulation Symposium.*

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. Parallel and Sequential priority Queue Algorithms. *ACM Trans. On Modeling and Computer Simulation 2, 157-209*.

Vaucher, J. G., and Duval, P. 1975. A comparison of simulation event lists. *Commun. ACM 18, 4(June), 223-230*.

AUTHOR BIOGRAPHIES

TAN KAH LEONG is a Research Scholar in the Department of Electrical and Computer Engineering, National University of Singapore (NUS). He received his B.Eng from NUS. His research interests include O-O simulation and neural networks. He can be contacted at <engp9186@nus.edu.sg>.

DR THNG LI- JIN, IAN is a lecturer in the Department of Electrical and Computer Engineering, National University of Singapore. His research interests include O-O simulation, signal processing and communications. He can be contacted at <eletlj@nus.edu.sg>.