SNOOPY A NEW CALENDAR QUEUE STRUCTURE

Tan Kah Leong
Dr Ian Thng Li-Jin
Dr Ko Chi Chung

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

Keywords: Optimization, Calendar Queue, SNOOPy CQ, Future Event List

ABSTRACT

Discrete event simulations need a priority queue to schedule events according to their timestamp. Up to 40% of the time may be spent on the management of the pending event set. Thus the choice of a efficient data structure is vital to the performance of discrete event simulations. A conventional Calendar Queue (CQ) and Dynamic Calendar Queue (DCQ) are two data structures that promises O(1) time complexity regardless of the size of the pending event set. CQ however performs poorly over skewed event distributions or changes in the event distribution. DCQ improves on the CQ by adding a mechanism to detect the two scenarios above and redistribute events when necessary. Both of these data structures determine their operating parameter (bucket width) by sampling of events. Sampling sometimes fails to provide a good estimate for an optimum bucket width to use. This paper proposes a novel approach to determine the optimum operating parameter of a calendar queue based on its performance statistics and guarantees O(1) performance. Henceforth, we named our calendar queue adopting this mechanism, Statistically eNhanced with Optimum Operating Parameter Calendar Queue (SNOOPy CQ). Our experiment results showed that SNOOPy CQ offers a consistent O(1) performance and, in certain scenarios, executes up to 100 times faster than DCQ and CQ.

1 INTRODUCTION

Discrete event simulations are widely used in many research areas to model 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 service 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 for a fine-grain simulation, it has been shown that up to 40% of the simulation execution time maybe spent on the management of the PES alone [Comfort, 1984].

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

A Dynamic Calendar Queue (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 matrices 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 describe the statistical data-bucket width relationship used by the SNOOPy CQ algorithm. Utilizing this relationship, section 4 describes the SNOOPy CQ mechanism in pseudocode. In section 5, the performance graphs of SNOOpy CQ, DCQ and CQ under different situations are presented, compared and analyzed. Finally section 6 summarizes the content of this paper.

2 CQ AND DCQ

A quick understanding on the working of the calendar queue can be obtained by following the illustrated example below.

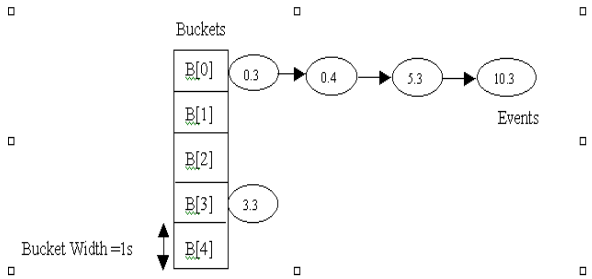


Figure 1: A Conventional Calendar Queue

A calendar can be represented by an array of buckets as shown in the figure above. Each bucket contains a linked list of events. Each bucket represents a single day of the calendar year. For the example above a year consists of five days. The length of a day is in turn represented by the bucket width and for this case it is 1 s. A year for the above calendar is thus 5 s. Events are sorted according to the day and year they fall into. Looking at the calendar above it can be seen that the events' timestamps can be mapped to the calendar year and day according to the Table 1

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

It can be seen that any events falling on the same calendar day regardless of their calendar year is inserted into the same bucket and sorted in increasing time order. All day one events are stored into bucket 0 or B[0], and all day N events are stored into bucket N-1 or B[N-1]. Let's assume we would like to insert a new event with timestamp 4.3 s. Mapping this timestamp to that of the calendar shows that it falls on day 5 and year 0 (4.3 modulo 5). Thus this new event will be inserted into B[4].

For dequeuing of events the calendar queue always keep track of the current calendar year and day it is in. It will dequeue the earliest events that falls on the current calendar year and calendar day. If no such event is found it will proceed on to search the next bucket. When all the buckets have been cycled through the current calendar year will be incremented by 1 and current calendar day will be reset back to day 1(B[0]).

The number of buckets of a calendar is of the power of two. Buckets are doubled or halved everytime the number of events exceeds twice that or decreased below half the current number of buckets. During this resize operation a new value of operating parameter (bucket width) will be calculated. The new bucket width to used will be estimated from sampling the average inter-event time gap from the first few hundreds events. With this new bucket width a new calendar will be created and all the events in the old calendar will be recopied over.

The resize heuristic described above for the calendar queue suffers from the following problems:

1) Since resizing is done only when the number of events doubled or halved that of the current number of buckets, calendar queue would not be able to adapt itself if there is a drastic change in event distribution but not on the number of events.
2) Sampling the first few hundreds of events for estimation of an appropriate bucket width to use may not give an accurate estimate for the optimum bucket width especially when event distribution is highly skewed.

Dynamic Calendar Queue (DCQ), tries to improved on the conventional calendar queue by adding the following mechanism. DCQ keeps track of the Average Enqueue Cost (AEC) and the Average Dequeue Cost (ADC). AEC is defined as the average number of events that need to be skipped through before an insertion can be made on a

linked list. ADC is defined as the average number of buckets that need to be searched through before the event with the earliest timestamp can be found.

DCQ computes AEC and ADC after every N enqueue or dequeue operations, where N is the number of buckets in the calendar. If the AEC or ADC computed exceeds a preset threshold indicating a change in event distribution, DCQ forces a resize operation. A new bucket width will be computed based on sampling of events around the most populated bucket. Although DCQ can detects and forces a resize operation when an uneven distribution of events in its buckets is detected, DCQ still suffers from the following problems:

1) The DCQ heuristic suffers from inherent instability and may perform better than conventional CQ in same situation but performs worse in others.
2) Calculation for of an optimum bucket width to use just by sampling events around the most populated bucket does not guarantee a good estimate everytime. It can give a better estimate then the estimation method used by conventional CQ in some situation but not all. This leads to the instability problem described in (1).
3) When both AEC and ADC does not exceed the threshold no further optimization of bucket width will be done although possible.

SNOOPy CQ uses a novel approach in estimating an optimum bucket width to use without resorting to sampling of events. It also incorporates an algorithm that does further optimization of bucket width from time to time to achieve optimum performance. The statistics behind obtaining an optimum bucket width for a calendar queue will be described in the next section.

3 STATISTICS BEHIND SNOOPY CQ

Performance of a priority queue can often be measured by the average number of searches performed for a set of enqueue and dequeue operations (a hold operation). Thus to optimize a calendar queue we will need to minimize ADC+AEC. Let's define the average searches done for a hold operation as Average Total Cost (ATC). Assume that the current ATC, ADC and AEC for a calendar queue is *Oldcost*, C_D and C_E respectively. And assume that currently $C_D > C_E$. Let the current bucket width of the calendar queue be *Oldbucketwidth*. Thus we have,

$$Oldcost = C_D + C_E \quad (1)$$

Our objective is to find a new bucket width, *Newbucketwidth*, that minimizes expression (1). Let the optimum resize factor be *k* such that

$$Newbucketwidth = k \times Oldbucketwidth \quad (2)$$

And this factor k minimizes expression (1) above. Since $C_D > C_E$, the calendar queue at the moment makes more searches through empty buckets before dequeuing an event than searches through the linked list for enqueueing an event. Thus to rectify this situation *k* must be bigger than 1. In other words we must increase the bucket size to decrease the number of empty buckets and effectively decreasing C_D. Let's assume that when we increase the bucket width by a factor of *k*, C_D will decrease by a similar factor *k*. AEC will also be affected by this resizing. If we increase the calendar bucket width by a factor *k*, more events will reside in each bucket, thus increasing AEC. Let's assume C_E will be increased by a factor *k*, when bucket width is increased by a similar factor. Table 2 shows the effect on the calendar variables before and after the resize operation.

Table 2: Variable Values for Calendar Queue

Variables	Before resize	After resize
Bucket Width	*Oldbucketwidth*	$k \times Oldbucketwidth$
ADC	C_D	C_D / k
AEC	C_E	$k \times C_E$
ATC	$C_D + C_E$	$C_D / k + k \times C_E$

From the table above we propose the new ATC, *Newcost*, as a function of resize factor *k* as

$$Newcost = C_D / k + k \times C_E \quad (3)$$

By differentiating (3) with respect to *k*, and equating it to zero, we find that the optimum *k* that minimizes the ATC is given in the equation below

$$k = (C_D / C_E)^{0.5} \quad (4)$$

Substituting (4) into (2), we obtain the expression below that gives an estimate of an optimum bucket width to use that minimizes the ATC.

$$Newbucketwidth = (C_D / C_E)^{0.5} \times Oldbucketwidth \quad (5)$$

For the case $C_E > C_D$, using similar arguments, expression (6) below can be derived for the optimum new bucket width.

$$Newbucketwidth = (C_D / C_E)^{0.5} \times Oldbucketwidth \quad (6)$$

Thus, the same equation can be used in order to obtain an estimate for an optimum bucket width to use for both situa-

tions. Substituting this value of *k* into Table 2 we obtained the following values for the calendar variables in Table 3.

Table 3: Variable Values for Calendar Queue

Variables	Before resize	After resize
Bucket Width	*Oldbucket-width*	$(D/E)^{0.5} \times Oldbucketwidth$
ADC	C_D	$(C_D \times C_E)^{0.5}$
AEC	C_E	$(C_D \times C_E)^{0.5}$
ATC	$C_D + C_E$	$2 \times (C_D \times C_E)^{0.5}$

From Table 3 it can be observed that for optimum operation the value of ADC and AEC are the same.
As for the ATC, since,

$$(C_D{}^{0.5} + C_E{}^{0.5})^{2.} \geq 0$$
$$C_D + C_E + 2 \times (C_D \times C_E)^{0.5} \geq 0$$
$$2 \times (C_D \times C_E)^{0.5} \leq C_D + C_E \quad (8)$$

Equation (8) shows that the ATC after resizing will always be smaller or equal to that ATC before resizing. Thus this resize algorithm is stable and will eventually converge to the true optimum bucket width that minimizes ATC after a few resize operations even if estimates for ADC and AEC are not so accurate.

4 SNOOPY CQ

SNOOPy CQ is very similar to that of DCQ except for the algorithm used for estimation of bucket width. SNOOPy CQ computes a new bucket width to be used based solely on the current bucket width and the performance statistics ADC and AEC. Initially when queue size is small and performance statistic is not available SNOOPy CQ relies on the DCQ algorithm to compute bucket width. Only when the queue size exceeds a certain threshold does SNOOPy CQ takes over the task of computing bucket width from DCQ.

It has been shown in section 3, that optimum performance occurs when ADC is same as AEC. To achieve this, SNOOPy CQ initiates performance fine-tuning even when ADC or AEC differs significantly from each other. SNOOPy CQ will initiate a bucket resize operation whenever the ratio of a 10 samples moving average of ADC/AEC differs by more than a factor of two (MADC> 2 × MAEC or MADC < 0.5 × MAEC). The moving averages, MAEC and MADC, is the average for the 10 most recent values of AEC or ADC respectively. This algorithm ensures that both ADC and AEC will have approximately the same value for optimum operation.

The pseudo-code for SNOOPy CQ for an enqueue operation is given in Figure 4 and for the resize operation in Figure 5. A description of all the variables used is given in Table 4 below.

Table 4: Description of Variables

Variable	Description
CalQSize	Keeps track the number of enqueued events
CalTop-Threshold	The threshold for upward resizing
EnqCnt	Keeps track of the number of events skipped before a successful insertion into a linked list
SumEnqCnt	Accumulated EnqCnt since the last resize operation
EnqSearch	The number of enqueue operations done since the last resize operation
DeqCnt	Keeps track of the number of buckets skipped before a successful removal of the most imminent event
SumDeqCnt	Accumulated DeqCnt since the last resize operation
DeqSearch	The number of dequeue operations done since the last resize operation
AEC	SumEnqCnt/EnqSearch
ADC	SumDeqCnt/DeqSearch
MAEC	Average value for 10 most recent AECs
MADC	Average value for 10 most recent ADCs

```
enqueue( ) {
(1)enqueue new event to the appropriate bucket;
(2)  set  EnqCnt  to  the  number  of  events  to
visit;
(3) calQSize++;
(4) if (calQSize > calTopThreshold) {
(5)  calResize(2 * calNBuckets);
(6)  SumEnqCnt = 0;
(7)  EnqSearch = 0;
(8) }else {
(9)    SumEnqueueCnt += EnqCnt;
(10)  EnqSearch++;
(11)  if (EnqSearch > calNBuckets) {
(12)     if((SumEnqCnt/EnqSearch>2)
         ||(MAEC/MADC>2)| |(MAEC/MADC<0.5)){
(13)             AEC=MAEC;ADC=MADC;
(14)         calResize(calNBuckets);}
(15)     else{
(16)     Collect Moving Average of AEC;
(17)        EnqSearch = 0;
(18)        SumEnqCnt = 0; } } } }
```

Figure 2: Enqueue() Pseudo Code of SNOOPy CQ

Line (3) to (8) shows the static resize algorithm of a conventional CQ that resizes only when calQSize exceeds

twice that of the current number of buckets. The dynamic resize algorithm of SNOOPy CQ is given by line (9) to (18). SNOOPy CQ dynamic resize algorithm is very similar to that of DCQ except on line (12) and (16). On line (12), SNOOPy CQ adds additional condition for resize whenever the ratio of 10 Samples Moving Average for AEC and ADC differs by more than a factor of 2. On line (16), moving average data of AEC is collected and this is absent in the DCQ algorithm. The effect of line (12) and (16) is to further fine tune the bucket width for optimum performance during steady state by ensuring that AEC and ADC does not differ much from one another.

The pseudo code for the calendar resize operation calResize( ) is given in Figure 3 below.

```
calResize(N){
(1)if( EnqSearch>64 && DeqSearch>64 )
(2)     bucketwidth = (ADC/AEC)^0.5bucketwidth;
(3)else
(4)  calculate bucketwidth using DCQ algo;
(5)reset MAEC and MADC collected;
(6) redistribute events using the new bucket-
width calculated;
  }
```

Figure 3: Enqueue() Pseudo Code of SNOOPy CQ

From figure 3, it can be seen that when the number of statistical samples are big enough to get an accurate estimate for AEC and ADC then only SNOOPy CQ bucket width estimation algorithm is triggered. If there is not sufficient amount of data to compute a good estimate of AEC or ADC the standard DCQ bucket width estimate algorithm will be used. Line (5) resets the MAEC and MADC obtained since a resize operation changes the *bucketwidth*. MAEC and MADC measure the moving average of AEC and ADC for at constant *bucketwidth*.

5 EXPERIMENTS AND RESULTS

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 4.

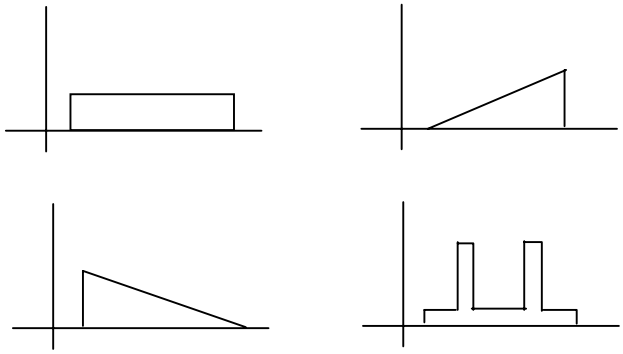
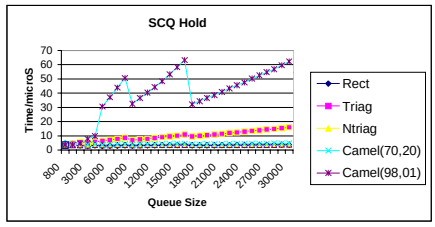
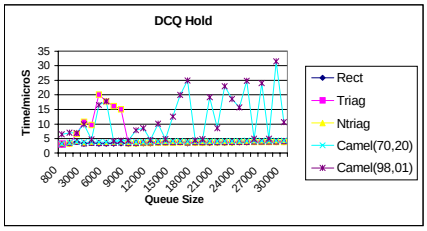


Figure 4: Benchmarking Distributions

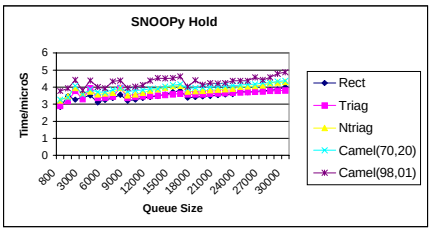
The Classical Hold and Up/Down model represent two extreme cases and are frequently used to show the performance bounds of PES implementations [Vaucher and Duval, 1975]. The number of hold operations performed is 100 × the queue size. Loop overhead time is eliminated using another dummy loop as was described by Rönngren and Ayani[1997].. Figure 5 shows the Hold results under different distribution for SCQ, DCQ and SNOOPy CQ.



(a)



(b)

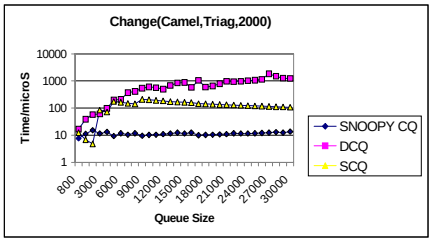


(c)

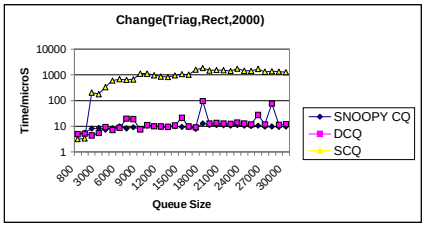
Figure 5: Average time per Hold operation for SCQ, DCQ and SNOOPy CQ

It can be observed that out of the three queue implementations SNOOPy CQ seems to be least affected by the type of distribution used with average hold time between 3 to 5 µs for all priority increment distributions. DCQ performance is erratic especially for Triag and Camel(98,01) distributions. Average hold time varies from 3 to 30 µs. SCQ performance is the worst among the three queue implementations with average queue time varying from 3 to 65µs. It is most affected by the Triag and Camel(98,01) distributions. Both DCQ and SCQ suffer from the same problem in estimating the optimum bucket width to use just from sampling of events. For DCQ sampling of events around the most populated bucket seems to give a good estimate for some situation but not all. Thus, the inconsistent performance as shown in graph (b).

Two other distributions used for the Hold benchmark are the Change (camel9801(9-10), Triag(0-0.0001), 2000) and 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 Tiag(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 benchmark are shown in Figure 6.



(a)

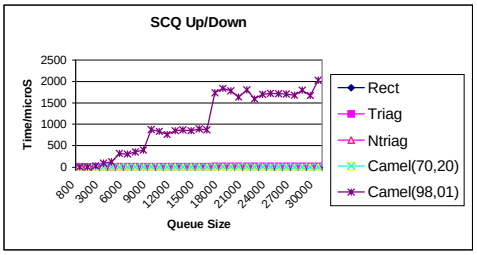


(b)

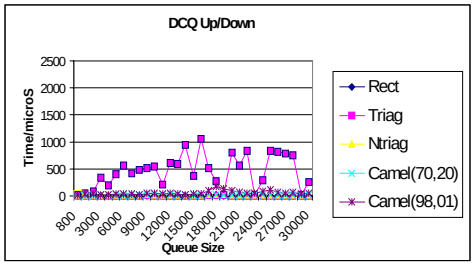
Figure 6: Average time per Hold operation under Change(A,B,x)

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 (a) and (b). The resize hueristics for SCQ and DCQ fails 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. SCQ 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 sampling of events does not offer consistent performance under all situations unlike the SNOOPy CQ resize heuristic.

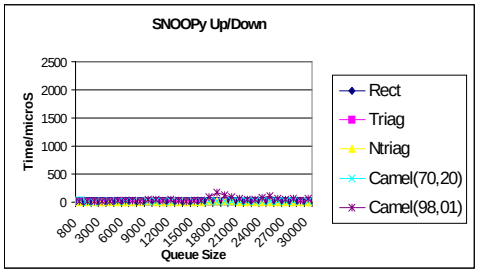
For the Up/Down model a total of 10 cycle of filling up the Calendar up to the required queue size followed by complete emptying of the calendar was done. The average time per Hold operation is then computed and plotted against different queue sizes. The plots for SCQ, DCQ and SNOOPy CQ under different priority increment distributions are given in Figure 7.



(a)



(b)



(c)

Figure 7: Average time per Hold operation under Up/Down Model

Graph (a) shows that SCQ resize heuristic is sensitive towards Camel(98,01) distribution although resize operations happen often enough. This is due to the fact that SCQ

determines the bucket width inaccurately by just sampling the first few events.

Graph(b) shows that the DCQ resize heuristic is works well under most distributions except Triag. The heuristic tends to estimate a bucket width that is too small just by sampling events around the most populated bucket.

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

6 CONCLUSION

Choosing the correct data structure to use to implement a PES for a simulator is important to speed up simulations when simulations generate huge number of events. 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). The statistical aspect behind SNOOPy CQ has been explained and implementation of it in pseudo-code was outlined. Experimental results from the Hold and Up/Down model shows that SNOOPy CQ consistently offers O(1) processing complexity under a wide selection of different distributions, unlike SCQ and DCQ. In the worst-case scenario SCQ and DCQ are 100 times slower than SNOOPy CQ, while in the best-case scenario their performance is of similar order with SNOOPy CQ.

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>.

A/P KO CHI CHUNG is the Deputy Head of Research and a lecturer in the Department of Electrical Engineering, National University of Singapore. His research interests include signal processing and communications. He can be contacted at <elekocc@nus.edu.sg>. His homepage is located at http://www.ee.nus.edu.sg/ee/view1.asp?user=elekocc