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 $0 \le k \le N_B\text{-}1$

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.

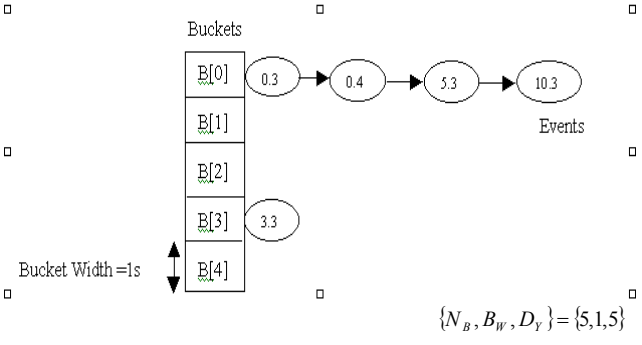


Figure 1: A Conventional Calendar Queue

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.

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

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:

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

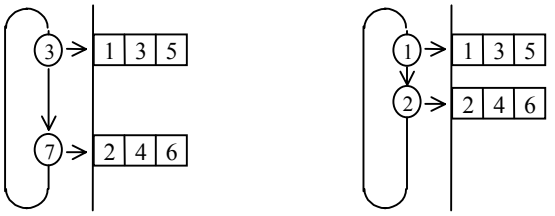
Note that the new average cost metrics C_D' and C_E' may remain optimized only for that short time period immediately after the bucket width upsize event has occurred, i.e queue distributions has not changed much before and after the upsize event. To handle a growing or declining PES scenario, more such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

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

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

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

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