Smart Priority Queue Algorithms for Self-Optimizing Event Storage

H. A. Bahr and R. F. DeMara

Department of Electrical and Computer Engineering
University of Central Florida
P.O. Box 162450
Orlando, FL 32816-2450
Tel: (407) 823-5916 Fax: (407) 823-5835
e-mail: hab@hbahr.org

ABSTRACT

Low run-time overhead, self-adapting storage policies for priority queues called Smart Priority Queue (SPQ) techniques are developed and evaluated. The proposed SPQ policies employ a low-complexity linear queue for near-head activities and a rapid-indexing variable bin-width calendar queue for distant events. The SPQ configuration is determined by monitoring queue access behavior using cost-scoring factors and then applying heuristics to adjust the organization of the underlying data structures. We show that optimizing storage to the spatial distribution of queue access can decrease HOLD operation cost between 25% and 250% over existing algorithms such as calendar queues. An SPQ-based scheduler for discrete event simulation has been implemented and was used to evaluate the resulting efficiency, components of access time, and queue usage distributions of the existing and proposed algorithms.

Keyword: Discrete event simulation; Priority queue; Adaptive algorithm

1.0 Introduction

We present Smart Priority Queue (SPQ) policies for inserting, deleting, retrieving items in the event queues of discrete event simulators. A fundamental capability required is an efficient means of storing and selecting the events contained in the process queue. The SPQ techniques decrease average access overhead by selecting a more efficient storage structure for the particular distribution of events encountered during simulation. Rather than specifying a single management scheme for the priority queue of simulator events, the SPQ approach dynamically selects between

structures such as a linear queue for near-term events and a calendar queue for more distant events.

1.1 Role of Event Management in Discrete Event Simulation

During discrete event simulation, time is considered to advance between a sequence of explicit events which occur at discrete instants. Although the management of time-flow in the simulation can be handled either synchronously or asynchronously [4], the asynchronous scheduling is often preferred [5]. In this case, a prioritized list of future events must be maintained so that individual tasks can be scheduled for execution at the appropriate time.

The data structure used for asynchronous time management in the simulator is the *event list* of activities scheduled for simulation. The event list is organized as a priority queue with time as the basis of prioritization for the position of the elements within the queue. Early simulation libraries used a linear linked list algorithm while more recent tools such as CSIM and YACSIM [9], use more sophisticated algorithms such as the calendar queue [3] to improve performance. The linear linked list algorithms, which store the events sequentially, have the disadvantage of requiring a search from the beginning of the queue to determine the proper location in the queue. Since each step in a simulation frequently involves scheduling new events for future execution, this has the potential for dominating overhead when large queues are required. The calendar queue technique improves performance by bounding the number of comparisons, but introduces additional overhead for resizing of bins during execution. In practice, however, the frequency of insertions in a small sub-range of the queue in some simulations allows the linear list to outperform the calendar queue.

A standard metric for comparison of the relative performance of an event list management algorithm is the time required for a HOLD operation [10][3]. A HOLD operation consists of first retrieving the event at the head of the scheduling queue and then inserting a new event into the appropriately prioritized location. Measuring time spent to execute HOLD operations provides a representative measure of event list overhead during simulation by exercising queue insertion and removal. Another fundamental task used to assess performance is DELETE operation which removes a superseded event which is no longer pending. Supporting arbitrary DELETE operations can cause performance degradation for some priority queue strategies such as heaps.

1.2 Motivation for Optimizing Queue Algorithms in Discrete Event Simulation

During simulation of a large scale discrete event simulation model that had an average queue size more than 2,000 events, it was observed that a linear linked list outperformed a calendar queue [2]. An investigation was initiated to determine the reason for the anomaly and to find potential optimizations for managing the event list. As previously suggested by Gonnet [7], specific distributions of non-uniform events can occur during simulation that can impact the performance of the event list storage strategy. Likewise, Brown observed that queue statistics should be continually monitored to determine which storage structure will minimize overhead for the particular distribution of events encountered [3]. This paper investigates queue usage in a large-scale simulation and presents a new priority queue algorithm that provides improved event list performance.

1.3 Outline of Paper

Section 2 describes previous work with respect to comparison of queuing strategies suitable for discrete event simulation. Section 3 presents event list statistics of the simulation application used for algorithm comparisons. Section 4 presents the design of the SPQ algorithms along with the rationale behind their design. Section 5 presents performance results. Section 6 presents conclusions and a discussion on applicability to other applications.

2.0 Literature Review

Significant literature on the design of event lists has concentrated on efficient implementations of the priority queue. In a priority queue the ordered structure of the elements must be continuously maintained according to their relative time to be scheduled during the

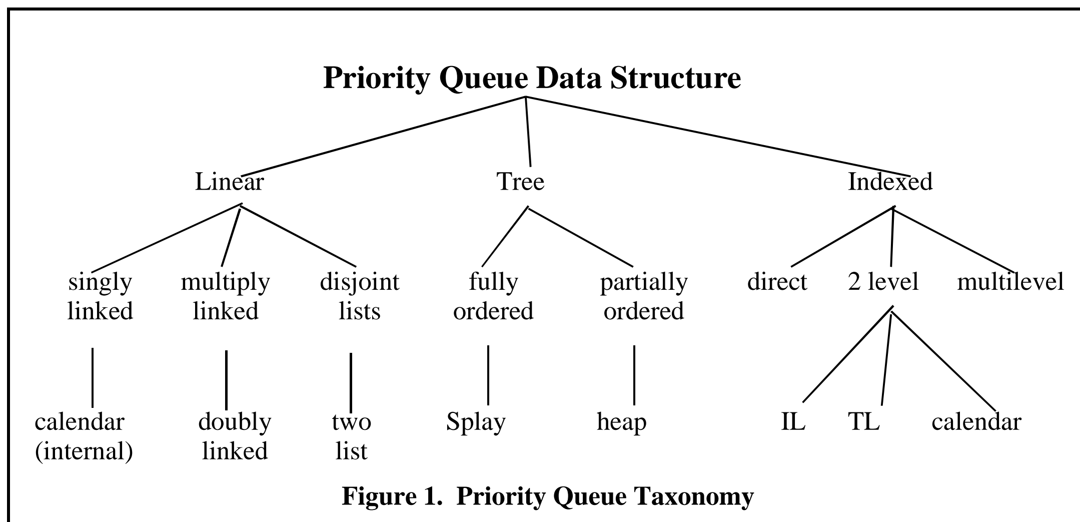

Figure 1. Priority Queue Taxonomy

simulation. For elements of equal priority, an accepted practice is to store them in First-In First-Out order. Figure 1 illustrates the various structures that have been explored for implementing priority queues. We will focus on linear and indexed data structures as they form the basis for the SPQ techniques.

2.1 Linear List Priority Queues

Linear lists can be divided into subcategories of singly, multiply, and disjoint lists [8]. The linear queue is formed by maintaining pointers to the head and tail of a singly-linked list. Figure 3 illustrates a singly-linked linear queue containing an initial list of events to be scheduled at times 0.2, 0.4, 0.5, 0.9, 1.2, 3.5, and 7.2. During an enqueuing operation, the events are stored in their prioritized location by repeatedly comparing the simulation time of the new event to the previously stored values. A new entry has its priority compared first with the priority of the head element and if it is less than the head then it is immediately inserted as the new head as shown in Figure 2. Otherwise, it is compared to the tail. If it is greater than or equal to the tail then it is appended as the new tail of the list. If it is neither a new head nor a tail, the priority is compared in

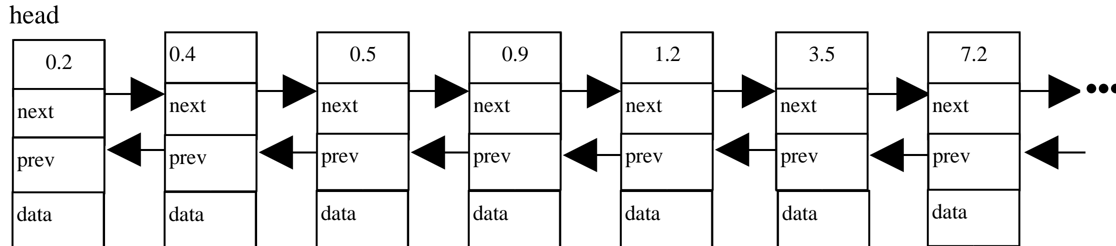

Figure 3. Linear List

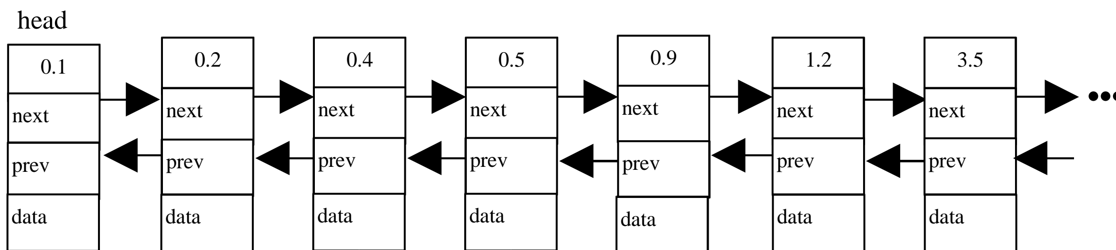

Figure 2. Linear List after HOLD at TIME = 0.1

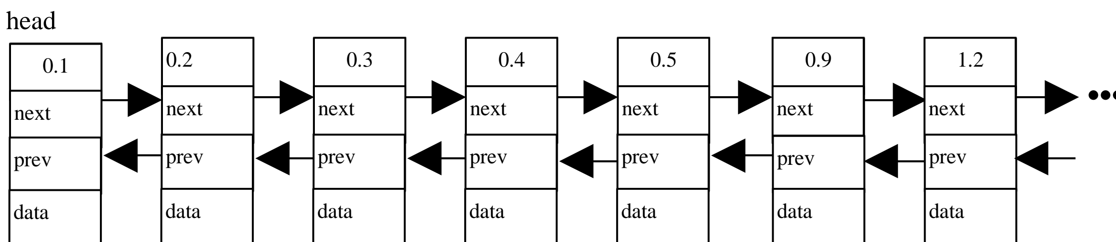

Figure 4. Linear List after HOLD at TIME = 0.3

succession with each next item in the list until it is less than the succeeding entry. Upon locating the correct point, the new entry is inserted in the list by updating the corresponding pointers. For example, insertion of the event a with priority of 0.3 in Figure 4. The primary advantage of this queue structure is that head of the list is always available without searching. Insertion and removal operations generally require two pointer updates. The disadvantage is that successive entries must be examined until the correct entry is found.

2.2 Indexed Lists

Indexed lists attempt to mitigate the drawbacks of linear queues by reducing the amount of searching required to locate or insert an item. Voucher and Duval [10] introduced a time mapping algorithm called the *indexed list algorithm.* This algorithm consisted of a linear list for event storage and an array of pointers to a set of dummy markers inserted into the list with the last pointing to the overflow area as shown in Figure 5. It depicts the operation of the indexed list algorithm showing the simplified version with array of three pointers and an overflow area. The interval, DT, is set to 1. The notices with "X" are dummies. The markers are a fixed-time interval apart and as the current simulation time passes them they are moved forward into the overflow area and inserted into the list at the appropriate time-ordered point. In this implementation, time is represented by an integer variable, and no finer subdivisions are allowed. The time at which an event is scheduled can be used as an index to select from the list into which the notice must be placed. With a FIFO priority system, the new notice is placed at the end of the selected list, and no scan is necessary. Although this algorithm in its basic state must be generalized in order to be widely applicable, the basic idea of grouping is useful to reduce the scan-time. However, the

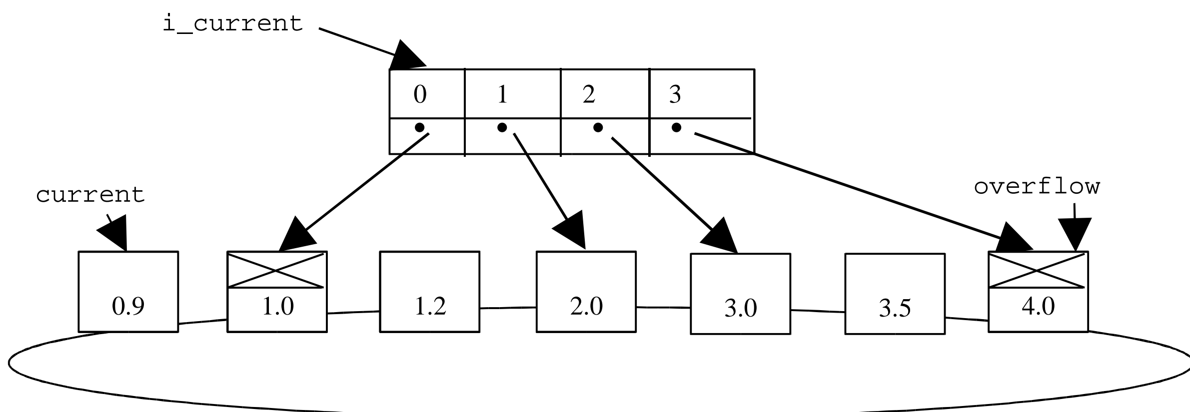

Figure 5. Indexed List Algorithm showing Initial conditions with four event notices. time = 0.5; lower_bound =0

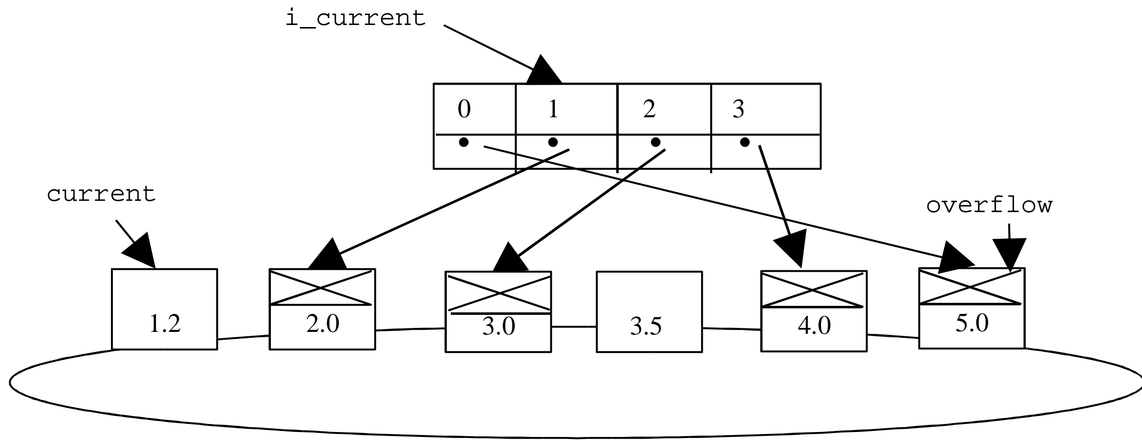

Figure 6. Indexed List Algorithm showing Conditions after HOLD (1.2). time =1.2; lower_bound = 1

implementation did not include a feedback mechanism for adjusting the spacing between markers. The working variables are `current` which provides a pointer to the current event notice, `i_current` which indicates the array pointer to the current interval and `lower_bound` which gives the simulated time for the beginning of the current interval. The last pointer in the array delimits the overflow portion of the list.

The operation of an index list is shown in Figure 5,6, and 7. As shown in Figure 5 for `time = 0.5` the `current` event is scheduled at 0.9 and the `lower bound` is `0.` The events scheduled at `times` 0.2 and 0.4 have already been executed. Figure 6, depicts a HOLD =1.2, where the `current` event is scheduled for 1.2 with a `lower_bound` of 1.0. As shown in Figure 7, for a HOLD at 3.2 the `current` event is scheduled for 3.5 with a `lower_bound` at 3.0.

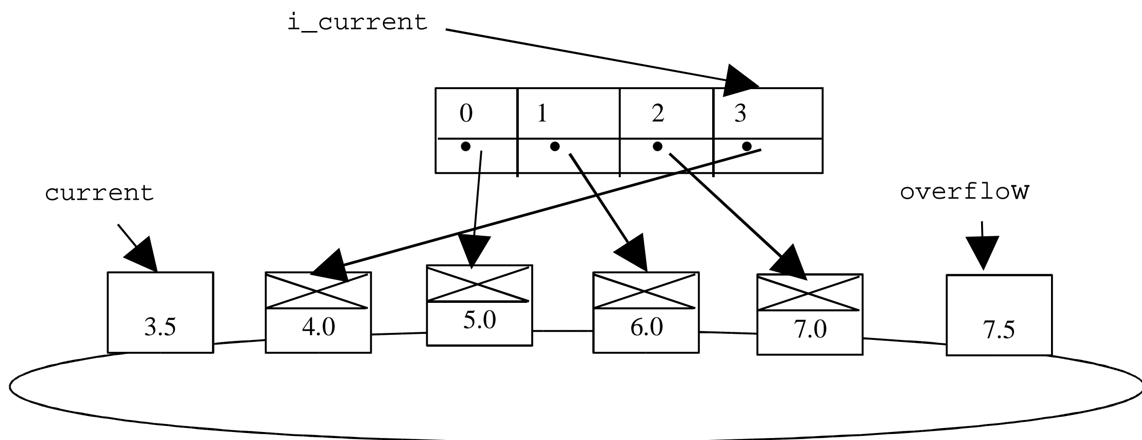

Figure 7. Indexed List Algorithm showing Conditions after HOLD (3.2). time =3.5; lower_bound = 3

2.3 The Calendar Queue

Brown [3] introduced the Calendar Queue, which derives its name from its structural similarities to a desk calendar. This is a multiple list structure, but offers many similarities to indexed list such as the TL structure of Franta [6]. However, it has simplified the indexing method and handling of overflows. The basic concept is that two arrays hold pointers to the head and tail elements of singly-linked lists of events. Each element of the list stores the priority of the event and a pointer to the next element. As shown in Figure 8, the length of the array is equivalent to the number of days in a calendar year. The figure depicts the queue with additional elements to illustrate storage in days 0 through 7. The index of each array is equivalent to the count of number of days since the beginning of the year, i.e. entry 0 is the start of the year and entry `n-1` is the last cell in the year where `n` is the number of days in the year. Overflow of events on any day is taken care of by placing the elements outside the current year, called *outyear* elements, in the appropriate day of the calendar in time sequence.

The index is calculated as *(priority/day_size) modulo year_size*, and converting to an integer value. If year sizes are kept as power of two, the modulo operation becomes a binary AND operation with a mask value of `n-1` where `n` is the year size. Interior to each day, the individual events are kept in priority order by scanning the list and inserting the elements in the appropriate location.

The calendar queue is implemented as a set of singly-linked lists where events are placed in time order. The correct list is determined by scaling the priority by the bin width W, and determining the bin number by taking the modulo of the queue length B. The resulting value forms the index into the two arrays that contain pointers to the head and tail elements of the appropriate lists. Overflow events are any events not in the current year of the calendar. Figure 8 illustrates a calendar queue with W equal to 1 and B equal to 8. During operation, the values W and B are adjusted to keep the number of elements in each list low. The key mechanism for calendar queue's improved performance over the linear queue is that it reduces the average sequential search length to half number of elements in a bin as compared to half the number of elements in the total list.

However, there is a price to pay for this capability, and it consists of several factors. The first is the basic bin selection process or indexing, this requires conceptually a floating point multiply, a floating point to integer conversion and then an integer modulo operation in addition to the compare and step, whereas, the linear search requires only comparison and advancing pointers. Another is the queue resizing cost, which requires sampling data in the queue, to calculate distance

Figure 8. Calendar Queue after insertions of Hold-0.1 and Hold=0.3

between events and moving the data to the appropriately spaced bins. The third is keeping track of the head element of the bin. In the linear queue, the head is always first in the list. In the calendar queue, there are as many list heads as there are bins. In a model with smooth evenly spaced events, the next head is always near to the current bin. In a worst case, it can lead to searching for the earliest event over the entire set of bins.

Brown identified several areas of concern that could cause variation in the expected performance of the calendar queue[3]. The first is that if the number of events in the queue is much smaller or much larger than the number of buckets, it will not function efficiently. He addressed this by implementing an auto-sizing mechanism which caused a readjustment whenever the queue sized changed by a power of two. Brown showed that this was bounded by the equivalent of 3 HOLDs in overhead cost. However, this allowed the possibility of oscillation

Figure 9. Smart Priority Queue (SPQ)

between the successive powers of two, i.e. successive enqueues to 2^n and then successive dequeues to 2^{n-1}. It is also possible for events to be clustered. This is a situation where the average space between events in a cluster is short, but have several years between clusters. He addressed this by testing such that if a full year was empty, it switched to search for earliest event, before normal enqueue/dequeue operations. Figure 9 depicts a new structure that addresses some of these concerns that will be dicussed in detail in section 4.

2.4 Relative Performance

Vaucher and Duval compared the performance of several algorithms which could be used to schedule events in a general purpose discrete simulation system [10]. Each algorithm is compared experimentally for performance of the HOLD operation over a range of queue sizes varying logarithmically from 1 to 200 elements. Table 1 presents the results of a previous study by Jones which compared average and worst-case performance of 11 different algorithms this showed performance advantages for splay trees over a wide range of conditions and linked lists (linear queue) for short lists [8]. Brown compared his calendar queue experimentally to the linear queue, and a queue implemented with a splay tree. The calendar queue consistently outperformed the splay tree for the scenarios tested. In fact, the calendar queue exhibited nearly constant-time performance for many queue sizes, while splay tree execution time increased O(log n) or in worst case linearly with queue size.

Priority-queue implementation	Code size[a]	Performance		Relative speed[b]	Comments
		Average	Worst		
Linked List	47	$O(n)$	$O(n)$	11	Best for $n < 10$
Implicit heap	72	$O(\log n)$	$O(\log n)$	8	
Leftist tree	79	$O(\log n)$	$O(\log n)$	9-10	
Two list	104	$O(n^{0.5})$	$O(n)$	9-10	Good for n < 200
Henriksen's	68	$O(n^{0.5})$	$O(n^{0.5})$[c]	1-7	Stable
Binomial queue	188	$O(\log n)$	$O(\log n)$	1-7	
Pagoda	110	$O(\log n)$	$O(n)$	4-8	Delete in $O(\log n)$
Skew heap, top down	56	$O(\log n)$	$O(\log n)$[c]	5-7	
Skew heap, bottom up	103	$O(\log n)$	$O(\log n)$[c]	4-6	Delete in $O(\log n)$
Splay tree	119	$O(\log n)$	$O(\log n)$[c]	1-3	Stable
Pairing heap	84	$O(\log n)$	$O(\log n)$[c]	3-6	Promote in $O(1)$

[a] The total lines of Pascal code for initqueue, emptyqueue, enqueue, and dequeue.
[b] 1 is fastest; 11 is slowest:
[c] An amortized bound; single operations may take $O(n)$ time!

Table 1. Results of Jones Study [8]

3.0 Utilizing the Event Distribution Characteristics

In many simulation systems, event activity is not uniformly distributed throughout the simulation time. Frequently the majority of activity occurs over a small interval at any time, for example near the beginning of the queue. This characteristic of the event scheduling distribution can be used to optimize queue storage structure. For example, consider a case study [1] involving simulation of a distributed communications system using YACSIM [9]. In this case study, the Range Data Measurement System (RDMS) for the U.S. Army was simulated where asynchronous communication between 2,000 source entities takes place over shared data channels to a central processing site. Several processors received the data in parallel and prepared it for retransmission in variously formatted data streams to other sites. To gain insight into placement and removal of activities on the event queue, several scenarios were simulated and statistics were gathered [2].

There were several classes of players each making different demands on the communications system and message length. There is also a process that generates random events that interrupt/initiate actions of these players. There is an additional process that monitors the loading of the communication channels and tries to keep the load balanced between the channels. These activities primarily influence the start of the message, but not the activity generated by the message. The reason the given message is delayed is because another message is being processed. On average each player initiates a message once every 5 seconds. Using a system time of 1 millisecond on there will be a new message every 2.5 time units on average, 5000 milliseconds/2000 messages. At this same time, that player will queue up to start a new message 5000 time units in the future. This message will occupy a communications channel for approximately 70 time periods and simultaneously interrupt the receiving processor for each byte transferred approximately 2.5 time units. This interrupt causes the processor to be used for 0.04 units of time. At the end of the communications channel time the message passes into the processor's where it goes through a long sequential series of 75 very short steps of 0.0002 milliseconds each which potentially conflict with data being received over one of the other communications channels or being prepared to go out. There is a shared common bus in the system that would tend to drive the ordering to be essentially sequential. The result is that on the average 1/75th of the events are queued near the end of the queue and 5/75 within 5 events of head of the queue and the balance at the head of the queue. Since the distribution of events was highly non-uniform, this lead to our study of self-adapting queue strategies used in the smart priority queue.

3.1 Evaluation of Queue performance for RDMS model.

The choice of data structure used to maintain the event list can significantly impact the efficiency of a simulation. In the case study of RDMS, despite its relative advantages, the calendar queue exhibited a cyclic nature and sensitivity to high head-end activity. The YACSIM library of simulation routines used to simulate RDMS did provide alternate strategies and evaluation tools that assisted in characterizing the problem. They provided optional queue implementations and debugging tools that allowed the capture of the queue at any instant. Figure 10 summarizes a bin snapshot as a histogram of fifty bins constituting 2.44% of the event queue, taken after the model being simulated has settled steady state. Bins 0 through 8 have only out year entries if any, while bin 9 has one last entry in the current year. Bin 10 has 47 current entries with one out year entry. The number of entries roll off to half size by 13 and empty by bin 49. A year for this queue is 4570 milliseconds, comprised of a W=2.23136 millisecond binwidth times 2048 bins, and the balance of the queue varies from 0 to a maximum of 5 entries with as many as 7 empty bins in succession.

A representative metric for the overhead required to support the distribution of event list activity is the number of data comparison operations for each event insertion. There were approximately 7.5 million insertions for one simulation run. Figure 11, shows the distribution of the baseline linear queue which had 4.5 million insertions at the head of the queue. The maximum search length was 2,301 and the majority of the events required more than 10 comparisons, however that the majority of events occurred near the head. Figure 12 shows the improvement offered by the baseline calendar queue in YACSIM with a maximum search length of 95

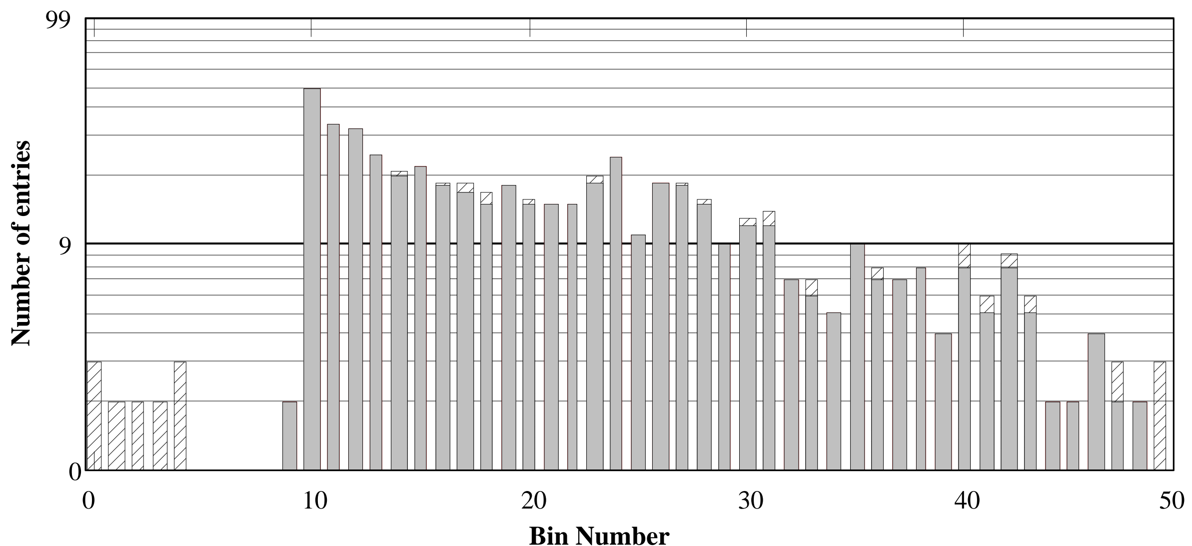

Figure 10. Histogram of Queue Snapshot

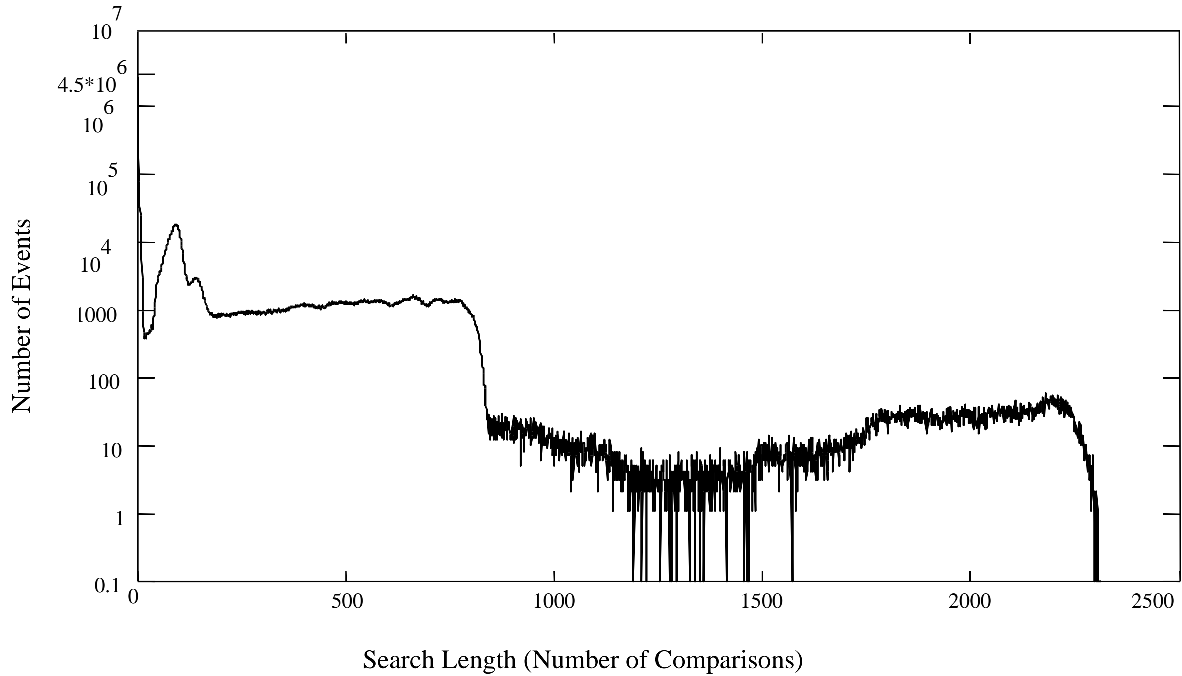

Figure 11. Distribution of search Length in DDC using a Linear Queue

comparisons. Thus, calendar queue effectively limits the maximum amount of time for any given operation, however the majority of the events are close to the head.

3.2 Model Analysis

The above data confirms that the majority of event list activity occurs near the head end of the queue. It also showed that a new headbin, the bin with the current head of the queue, would contain as many as 75 entries with another 75 entered and deleted during its lifetime. The bin at

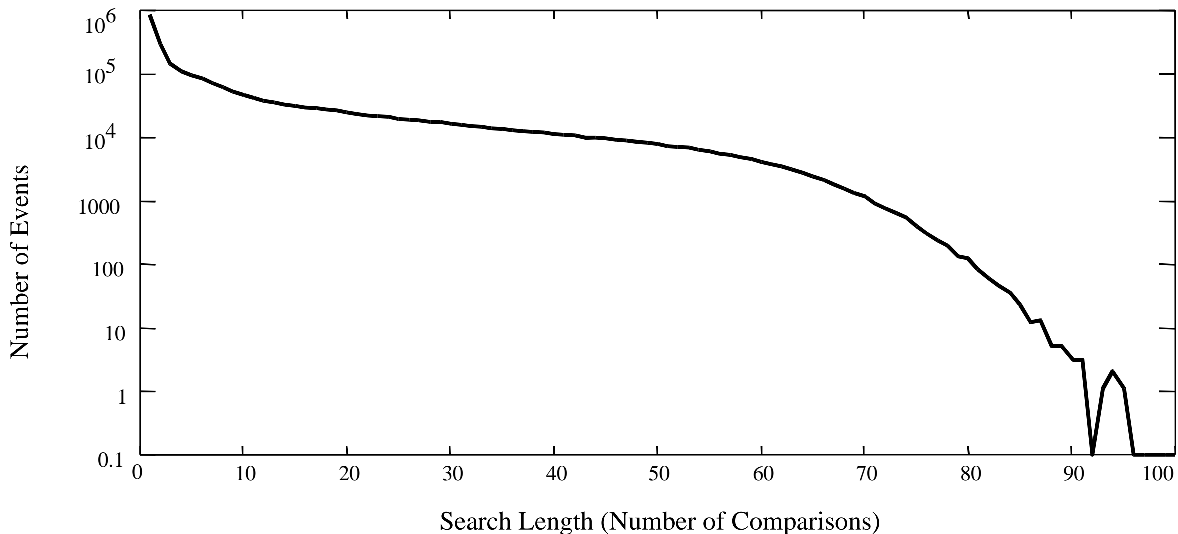

Figure 12. Search Length Distribution in a Calendar Queue

location headbin+1 was likely to have 45 entries. Since the default bin size is a fixed value of just 3 times the average event spacing calculated each time the length of the queue is adjusted, the model under simulation may reach a steady-state queue length, before it reaches a steady-state event distribution. Further analysis showed that all 2000 players are scheduled to start, and up to 120 additional events are scheduled, prior to the first increment in simulation time. In this situation, the model does not generate these closely spaced events until an external trigger is detected, the automatic width calculation in the calendar queue does not get the opportunity to execute because it never gets to see typical event spacing for this problem.

In summary, analysis and experimental measurements demonstrate scenarios with a high degree of head-end activity can occur after the queue size has reached steady state. Since a linear list has lower traversal overhead for near events, a simple linear queue can outperform the calendar queue for simulation with a heavy head-end activity.

4.0 Smart Priority Queue (SPQ) Data Structure

In order to accommodate non-uniform distribution of event list activity, several different structures based on modifications of the calendar queue were developed and evaluated. Once the benefits were fully understood, a distribution-adaptive data structure was developed to provide stable performance across a wide range of distributions.

The characteristics of the model emphasized that an optimal queue structure for event list management requires low overhead for the accesses that occur most frequently, such as head insertion and deletion. Furthermore, the structure should incur the minimum number of comparisons for all insertions. The list structure must be capable of accommodating rapid arbitrary deletes. In addition, the usage distribution can be difficult to describe analytically and its characteristics can change throughout a simulation run. However, a simulation library user should not be burdened with whether the selection of the appropriate priority queue structure for a particular simulation. Ideally, the priority queue would be close to optimum for all distributions. To satisfy these objectives, a dynamically-adaptable queue structure is required to consistently meet the following performance goals:

- minimize the total number of operations required to access the most frequently scheduled events,
- reduce the overhead cost of sample taking, resizing, finding the new head, by reducing their frequency of invocation,
- perform threshold testing only when beneficial, and

ï prevent oscillation in the storage structure during operation.

Nonetheless, adaptive mechanisms create the potential for oscillation. Although Brown identified the potential for oscillation, models used in his experiments did not excite this oscillation. While in an analog circuit, the performance cost of feedback is quite often ignored because parallelism is assumed among the feed-forward and feedback paths. However, in an adaptive algorithm, the cost of employing feedback during discrete event simulation directly increases simulation overhead, i.e. the same processor is employed for both the direct actions and the feedback operations.

Notably, the data structure with the least absolute operation cost is the linear singly-linked list. However, the linked list loses its performance advantage if its length exceeds about entries 10 [8], so an indexed structure such as a calendar queue is required to limit the list length and store items that overflow this length. Calendar queue performance can be adjusted by altering the width and number of bins. The primary disadvantage of the adaptive structure are the three components of overheads: sensing, evaluation, and adjustment. Their penalty can be reduced by minimizing the number of operations executed in the primary execution path and then amortizing high-cost routines such as adjustment over a large number of HOLD operations. The working of the SPQ can be explained using the Flowchart in Figure 13,

4.1 Activity Feedback Counters

The first three activity counters allow tracking of the individual paths and calculation of queue size and average insertion cost. `New_bin_count` allows the calculation of average number of `get-head` operations per `head_list` change. If this number is too low then it indicates that the bin-width is to narrow. The `head_over` and `bin_over` indicators are counts that give an indication of the `bin_width` being too wide as well as indicating an increase in the average number of compares per insertion. The number of these occurrences is more important than the number of compares per occurrence because each previous insertion that contributed to the length of the list, were also counted. `Bin_tail_over` is an indication that the calendar year is too short i.e., not enough bins. This also indicates an increased number of compare operations. `Skipped-count` is another indicator of narrow bins as well as an indicator of bursty data. It reflects the additional hold cost used in calculating optimal queue configuration. `Skipped_year_count` gives an indication of large gaps in data and has a cost proportional to the year-size or number of bins. This indicator identifies when the number of bins should be reduced.

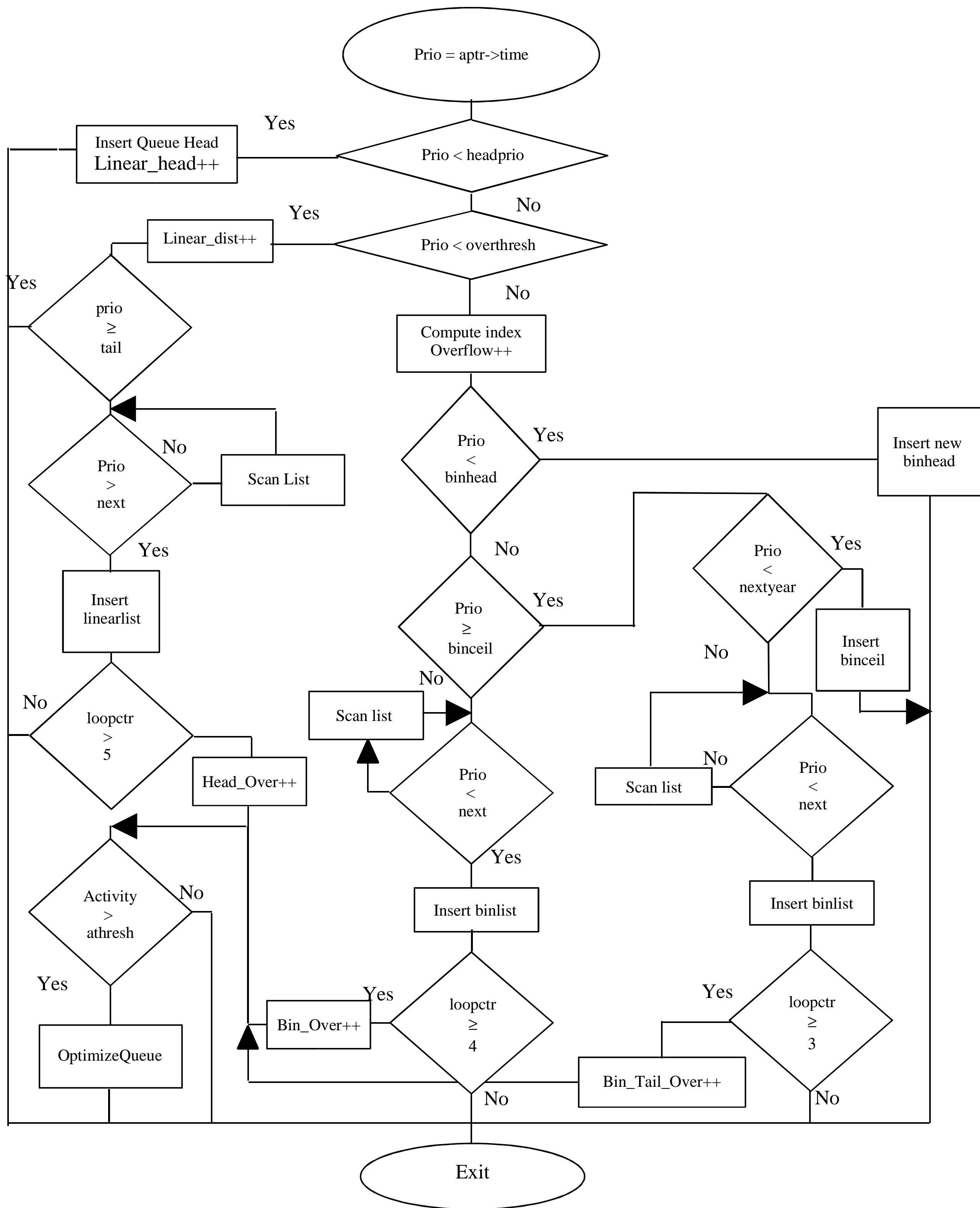

Figure 13. SPQ Insert Operation

4.1.1. Sensing Cost

The first cost of an adaptive algorithm is sensing. In Brown's case, he chose tracking of the queue size. For the SPQ case, since we also wanted to check other characteristics as well, the counts for each path were maintained separately. Different combinations of these counts could be used to determine queue size, activity, and where excessive operations occurred. Yet, we sought to keep the complexity of tracking information at the same level as Brown's. The other costs can be reduced by periodic sampling. The sampling strategy used is based on the observation that a change in the structure of the event list is not necessary until the distribution has substantially changed its characteristics. This is indicated by either excessive compares required for insertion, or excessive number of bins traversed between successive head lists. Therefore, insertion compare counts are monitored, and if they exceed a threshold, further tests are invoked. Likewise the number of bin changes are also monitored.

The SPQ is designed to minimize the total number of queue overhead operations by reducing the equivalent number of compares in the most frequently executed operations. The most frequent operation, other than examining the head of the queue, is removal of the head element. The overhead other than the counting operations over a linear list, is that the next pointer indicates a dead-end path more often. This signals the SPQ to transfer the following bin from the overflow structure to the head list. To optimize the queue, it is possible to trade off the frequency of changing the head list to the number of elements searched in the head list. For this purpose, the head list is initialized to contain 5 elements as a minimum to invoke this tax at most 20% of the time. The second factor is to minimize the number of operations required to move the list from the next bin to the head list. This was improved by having a pointer to the last element in the current year maintained during insertion. The result is to transfer the bin head and tail pointers to the head list and then setting the bin pointers to the head and tail of the next year. On the average the latter step requires one additional compare operation. Due to the high activity rates close to the head this adjustment, this was found to occur for less than 2% of the HOLD operations.

The next most frequent operation was insertion at the head of the queue. In this case, the SPQ behaves the same as the linear list with only one compare required besides the count. The next choice is to determine whether the new priority will be inserted in the head list or the overflow. Once the decision is made to place the event in the overflow structure, the SPQ operates very similar to the calendar queue. The result of using a separate head list is that for all elements stored in the calendar structure there are two compares, but over 50% of the insertions have been avoided. Bin hashing calculations and all head deletes are recouped in the bin indexing

operation. Another enhancement over the calendar queue is that the second pointer into each bin is not strictly a tail pointer, but rather a pointer to the head or the last entry in the current calendar year. This adds one additional compare for those events inserted over a calendar year away, but minimizes the number of compares required to transfer bin data to the head list. Insertion over a calendar year away occur rarely by the resizing design of the calendar queue.

Each list body insertion is monitored for the number of compares required to find the insertion point. If this exceeds a threshold value then further evaluation is initiated. The first test performed in the evaluation is to determine whether enough operations have occurred since the last restructuring of the queue for a new restructuring to provide potential benefit. This is a simple threshold comparison based on values calculated during the previous restructuring. This test serves as a damping function to insure the SPQ doesn't spend more operations adapting than it can save by restructuring.

4.1.2 Filtering Costs.

The second cost is analysis or filtering. In Brown's case it was a simple threshold test: if the queue size was greater or less than the thresholds, queue resizing was required. For SPQ, heuristics are used to first to determine whether a change is required and second what the parameter values should be for the resizing.

Six counters are used to monitor the performance. These are `linear_head`, `linear_dist`, `overflow`, `delete_count`, `get_count`, and `new_bin_count`. The first three of these reside in the separate branches taken in the priority queue for any activity. The last is incremented every time a bin is moved to the head list. This minimizes the cost of tracking the queue performance. All other monitoring is invoked only when an insertion occurs outside of the expected range. These are considered overflow conditions and are recorded by feedback counters for each of the following conditions:

ï `Head_over:` an insertion occurs in the head-list that takes over 6 compares.

ï `Bin_over:` an insertion occurs in a bin that takes over 4 compares.

ï `Bin_tail_over:` an insertion occurs on the end of bin-list that takes over 3 compares.

ï `Skip_count:` the number of empty bins skipped, and

ï `Skip_year_count:` the number of times a search had to be performed to find a new queue head.

When any overflow condition occurs, the first level of testing occurs. This first level is simply to determine if adequate activity has occurred in the queue to justify a change. It is a

threshold that is proportional to the queue size denoted by *N*. This test is a comparison of the sum of the activity counters to the threshold. The second test is to determine if the operational cost to make a change is less than the cost of allowing the overflows to continue. If the threshold is exceeded then optimizing calculations are made to determine the predicted queue parameters. These are filtered and compared to the current parameters and if the changes indicate improved queue performance the queue is adjusted. In all cases, the expected performance improvement must outweigh the operational cost of making the change.

4.1.3 Correction Costs

The final cost of feedback is correction. This is very expensive since it means setting up a new queue with corrected parameters then moving the contents from the old queue to the new one. Another consideration for feedback is response time, or how soon are adjustments made after the distribution changes. To make the queue more responsive if a change is required, the activity threshold is reduced to a value of 8 times the queue size from the nominal 100,000 operations. As long as changes are indicated then it remains at this level, once changes stop occurring the next check is performed at an interval equal to the number of operations since the last change. This has an effect of doubling the threshold until it exceeds 100,000, which is then used as the steady state sampling threshold. All heuristics have been grouped into one module for easier tailoring.

4.2 Performance Comparison

To establish a basis of comparison, it is possible to discuss cost of operations in terms of equivalent linear queue sizes. For most models the developer of the model can roughly estimate the equivalent linear queue length for the steady state operation. We intend to show that this

Operation	Cost (normalized instruction cycles)
Head_over	7
Bin_over	9
bin_tail_over	10
Skip_bin	6
New_bins	4
Search	3*B
Width	100
Resize	6*N

Table 2. Comparison Equivalent for Optimization Calculations

Queue Type	Execution Time (Seconds)	Variation in Execution time
Linear	1072	0.707 %
Calendar	696	1.917 %
Smart Priority Queue (SPQ)	544	1.399 %

Table 3. DDC Performance Comparison

implementation has a relatively small penalty for short queue sizes and is of order(1) after the average length exceeds the cost of the indexing operation.

5.0 Experimental Queue Comparison

Initial performance comparisons were based on total user time over the execution of the model for a repeated simulation run. Output data from each run was compared to verify that the model behaved identically for each run. This also verified that the enqueing technique did not change the FIFO ordering of events of the same time.

The results shown in Table 3 demonstrate the improvement gained from the changes to the queue implementation. Since the data listed in Table 3 indicates the wall clock execution time of

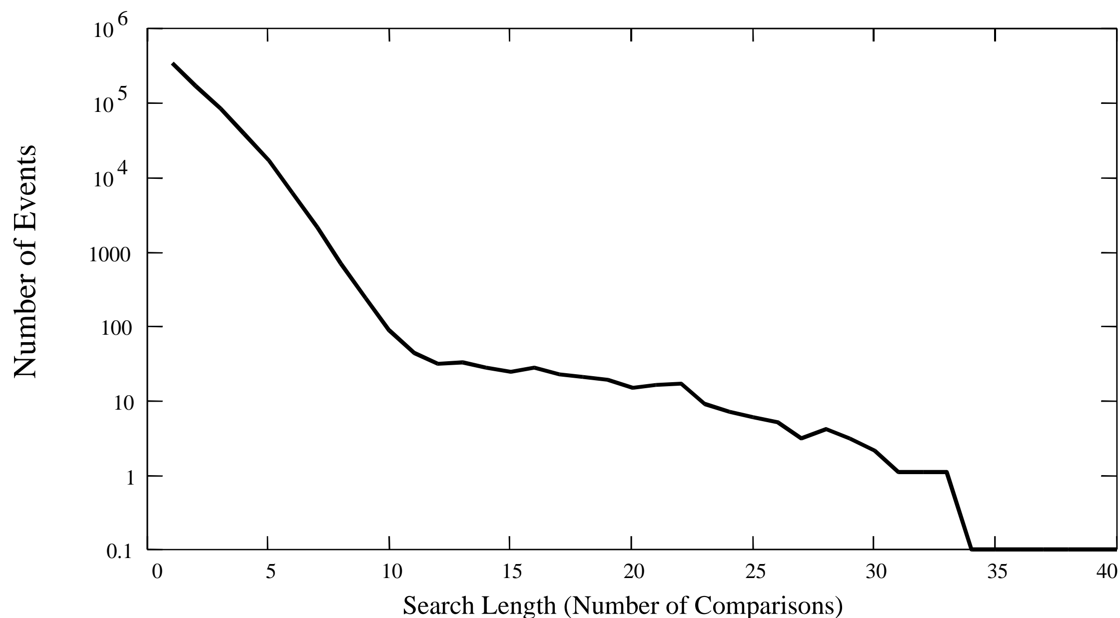

Figure 14. Search Length Distribution in a SPQ

the entire simulation, queuing overhead is just one component. Thus, a 21.8% reduction in wall clock time for the entire simulation by using SPQ rather than calendar queue is even more significant due to the changes is masked by the time required for the model to generate the data being enqueued. For this reason, and to further determine whether additional changes may be beneficial, the queue implementations were further instrumented.

5.1. Queue Instrumentation.

For each path through the queue maintenance routines, overhead statistic counters were added. When this path included a loop structure, the loop count was included. Counts for paths with no loops were accumulated and reported for the total simulation. Paths with loops had their results reported upon exit from the loop. The results are presented in Table 5. Figure 11, 12, and 14 provide the distributions in graphical form.

Counter	Description
Linear_head	The number of insertions at the head of a linear list
Index_head	The number of insertions at the head of a bin found after calculating the index.
New_headbin	The number of insertions at the head of the queue for the calendar queue and also part of the indexhead count.
Linear_tail	The number of insertions at the tail of a linear list.
Index_tail	The number of insertions at the tail of a bin found after calculating the index.
Index_empty	The number of insertions into a previously empty bin.
Linear_distrib	The number of insertions in the interior of a linear list. The average number of elements into the list where the insertion took place.
Index_distrib	The number of insertions in the interior of a calendar bin. The average number of elements into the bin where the insertion took place.
Get_head_linear	The number of element removals from the head of the linear queue as the event is activated.
Get_head_bin	The number of element removals from the head of the calendar queue as the event is activated.
Get_head_empty	The number of times the get_head action empties a bin.

Table 4. Statistical Counter definitions

5.2. Statistic counter definitions.

To collect statistics counters were used in various paths. These counters are only present if selected at compile time. Each counter is described in Table 4. There are 6 counters related to head-of-queue operations, 2 related to tail-of-queue operations, 2 related to the distribution, and 1 indicating the first addition to a bin.

5.3. Comparison of the results.

As shown in Table 5 the calendar queue outperforms the linear queue for the RDMS simulation by reducing the the number of comparisons (Figure 11, and 12) required to insert into the body of the queue. The SPQ further improves performance by taking advantage of the low overhead of the linear queue for the get head operation and further reducing the number of compares (Figure 14) to insert events in the body of the queue. Both the calendar and the SPQ queues introduce additional costs for monitoring and adjustment, but this is more than compensated for by the reduction in the number of compares required to order the queue.

OPERATION	LINEAR	CALENDAR	SPQ
Linear_head	4.52M		4.52M
Index_head		4.64M	678K
newheadbin		4.52M	
lineartail	879		265K
indextail		89.6K	494K
indexempty			222K
lineardistrib	2.82M		711K
indexdistrib		2.62M	445K
TOTAL INSERTIONS	7.35M	7.36M	7.35M
Total Index		7.36M	1.39M
Compare	319.28M	28.69M	1.78M
getheadlinear	7.35M		7.35M
getheadbin		7.35M	
TOTAL DELETIONS	7.35M	7.35M	7.35M
findheaddist ave		7.35M 1.01	14.9K 1.97
width samples			108
width change			11
resize change		10	10
TOTAL Compare Equivalent	359M	155M	59.0M

Table 5. Path Counts and Comparison Equivalents

6.0 Conclusion

Since since short hold time events can occur much more frequently is a simulation than long hold times, adaptive queue management techniques that capitalize on this characteristic can significantly reduce queue overhead.. Adaptive techniques were successfully applied by developing the SPQ to allow O(1) performance for these distributions. An important characteristic is low-overhead sensing of queue performance, which in turn triggered the adaptive measures required to bring the queue back within the optimal range of operation. A key benefit of these results is that they allow the prospective user to have confidence that the event queuing distribution will not drastically change the execution time of the simulation model.

SPQ performance whether analyzed from an operation count or a timing exercise, can show a reduction in overhead of greater than 50% in comparison to the calendar queue, and will perform no worse than the better of the linear queue or the calendar queue individually. Analysis of the SPQ shows that it will perform comparably to a linear queue for less than 8 events and will exhibit nearly constant time performance for larger queue sizes. SPQ performance can be shown to be better than that expected for binary queues with less complexity in the primary paths of execution. The SPQ algorithm has been used with several simulations and the source code is available from the author. Follow-on work includes tuning the heuristics for special cases where the dynamics of the distribution raise the average insertion cost appreciably.

References

[1] H. Bahr , Combined Event and Process Simulation Model of a Distributed Data Collection System, in Proceedings of the 1994 IEEE Southcon Conference, Orlando, Florida, (1994).

[2] H. Bahr, A Distribution Adaptive Priority Queue Algorithm for Discrete Event Simulation, Master Thesis, Electrical and Computer Engineering Department, University of Central Florida, Orlando, Florida, 1994.

[3] R. Brown, Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem,. Communications of the ACM, Volume 31, Number 10, October (1988).

[4] J. R. Emshoff, Jr., and R. L. Sisson, Design and Use of Computer Simulation Models, (MacMillan, New York 1970).

[5] G. W. Evans, G. F. Wallace, and G. L. Sutherland, Simulation Using Digital Computers, (Prentice-Hall, Englewood Cliffs, N.J., 1967).

[6] W.R. Franta and K. Maly, An Efficient Data Structure for the Simulation Event Set, Communications of the ACM, Volume 20, Number 8, August (1977).

[7] G. H. Gonnet, Heaps Applied to Event Driven Mechanisms, Communications of the ACM, Volume 19, Number 7, July (1976).

[8] D. W. Jones, An Empirical Comparison of Priority-queue and Event-set implementations, Communications of the ACM, Volume 29, Number 4, April (1986).

[9] J. R. Jump. ,YACSIM Reference Manual, Version 2.1 Rice University, March 1993

[10] J. G. Vaucher and P. Duval "A Comparison of Simulation Event List Algorithms,” Communications of the ACM, Volume 18, Number 4, April (1975).