A Performance Comparison of Event Calendar Algorithms: an Empirical Approach*

KEHSIUNG CHUNG, JANCHE SANG AND VERNON REGO Department of Computer Sciences, Purdue University, West Lafayette, IN 47907–1398, U.S.A.

SUMMARY

We propose a suite of tests based on two-state Markov chains for experimentally assessing the dynamic performance of a variety of simulation event calendar implementations. In contrast to previous studies based on the standard hold model for evaluation of performance statically, the proposed Markov hold model is more general and can be used to examine how different implementations respond dynamically to dependent sequences of insertion and deletion requests. The Markov hold model is used to conduct tests based on random, stressed, and correlated input sequences of requests, with performance measures including completion times, sensitivity to correlations, sensitivity to duplication, and efficiency of data-handling. We apply these tests to fourteen different event calendar implementations. To demonstrate the utility of the proposed model, we also include a comparison of the event calendar algorithms on a token ring protocol with bursty Markovian packet-traffic.

KEY WORDS: Event calendar Markov chain Simulation Hold model Priority queue

1. INTRODUCTION

Priority queues are well-known discrete structures in computer science applications with uses as varied as in the ordering of states in enumeration problems, optimization in search problems such as graph traversals, sorting problems and task scheduling in operating systems. The most popular use of priority queues is perhaps in the area of discrete event simulations; these are self-perpetuating programs which mimic random phenomena by processing previously scheduled events at discrete instants in time, and scheduling future events during such processing.

A simulation which begins with one or a few initially scheduled events ensures that as events occur in time, a sufficient number of new (future) events are generated so that program execution can continue indefinitely. The scheduling mechanism that effects this behavior does so by using a special structure, called the simulation *event calendar*,⁴ to store events which are supposed to occur at some future time. These pending events remain in the event calendar, in some data-structure dependent order, until the simulation program decides to extract an event at a time for processing. New events are scheduled and inserted into the calendar according to model specifi-

0038–0644/93/101107–32\$21.00 © 1993 by John Wiley & Sons, Ltd.

Received 3 August 1992 Revised 13 April 1993 cations, and these are usually generated while some related event is being processed. An event that has just been removed from the calendar is called the *current event* since its time coincides with the current simulation time; its processing is imminent.

After processing the current event and possibly scheduling some new events, a simulation program retrieves from the event calendar an event with the highest priority. Because of this, a priority queue can be used to implement the event calendar. The event with highest priority is typically one with the nearest scheduled time of occurrence, although in certain situations (e.g. pre-emptive service in queueing systems) an event with a larger scheduled time of occurrence may have higher priority. It is possible for two or more events to have the same priority; here the analyst is responsible for specifying how such ties are to be broken. An implementation which extracts such equal-priority events from the calendar in the same order in which they were inserted is said to be *stable*.³ We refine this notion by using the term *time-stable* to denote such implementations.

Since discrete event simulations tend to be compute-intensive applications, and since a significant fraction of the computation effort may be spent in event calendar processing,⁵ choosing a good event calendar algorithm is important. Past research has shown that event calendar processing can consume as much as 40 per cent of simulation time.^{6,7} McCormack and Sargent⁸ show that choice of event calendar algorithm can have a significant impact on simulation time.

In this paper, we introduce a technique to compare several well-known event calendar algorithms based on observed performance when subject to specific types of inputs. Hence our focus is on measuring the performance of an event calendar algorithm on the sequence of requests generated by a simulation. In this way, our measurements pertain solely to algorithm performance and are independent of machine-related effects arising during use of a simulation tool in conjunction with an algorithm. The latter is the topic of a sequel to this paper.

We introduce certain schemes for testing and assessing the dynamic behavior of event calendar algorithms. These are based on a simple but useful variant of the well-used *hold model*⁹ that we call the *Markov hold model*. Using this two-state Markov chain based model we conduct an empirical investigation of the dynamic behavior of fourteen different event calendar implementations. To demonstrate the utility of the Markovian model, we also compare the performance of these different implementations on the simulation of a token ring local area network protocol.

Our overall intent is to make this a thorough comparison of well-established event calendar algorithms. Since event calendars are often implemented using priority queues, most of our implementations are priority queue based. A few of these implementations, such as the Henriksen algorithm¹⁰ and the calendar queue¹¹ were developed specifically for event calendar processing; hence these simulation-specific algorithms may possess characteristics that are absent in priority queues. Henceforth, the terms algorithm, implementation and priority queue are all used to denote event calendar processing.

The remainder of this section contains a brief description of the standard hold model used in previous event calendar evaluations, a summary of prior empirical work, and finally our motivation for conducting this study. We also describe the Markov hold model along with the methodology used in the experiments of Section 2. This section contains three kinds of event calendar tests, along with experimental results for each kind. It also contains a description of the token ring simulation and

a comparative assessment of calendar implementations based on this application. The different event calendar implementations used in this paper are briefly described in the Appendix.

1.1. The standard hold model

The original Vaucher–Duval *hold model*⁹ was introduced solely to approximate a simulation application's interaction with its event calendar. An application is assumed to operate with a *fixed* number of initial events in the calendar. In operation, it *deletes* the earliest event from the calendar, synchronizes the simulation clock with this event's scheduled time of occurrence, processes the event, and then *inserts* a newly scheduled event into the calendar. In practice the hold model is implemented with precisely this sequence of operations, by ignoring all event processing and using an arbitrary, but fixed, scheduling distribution. For example, the control structure used by Jones¹² is shown in Figure 1.

The strictly alternating sequence of insert and delete operations clearly can be used to measure the average time taken to perform a hold operation, for a given calendar implementation. The question that remains is: does this adequately reflect what goes on in a simulation application? As eloquently argued by Evans,³ we believe the answer to this question to be in the negative. To reach this conclusion it is necessary to consider situations that can cause insert/delete sequences that are different from those used in the hold model, and gauge the effects of these sequences on algorithm performance. It is worth indicating that our goal is not an evaluation of the hold model. We are primarily concerned with comparing the performance of various event calendar algorithms under more general conditions that those of the hold model.

```
begin
  • Initialize event calendar with 1000 events;
  • done := false; hold_count := 0;

    Begin total_time measure;

  while (not done) do {
     Delete highest priority event from calendar;
     Synchronize clock with this event's time of occurence;
     Define a scheduled time to be clock time plus a random
       variate from a given scheduling distribution;
     Encode scheduled time in new_event;
     Insert new_event in calendar:
     hold_count := hold_count + 1;
      if (hold\_count = n) then done := true;
    } /* of while */
  • End of total_time measure
     hold_lime = total_time/n
end /* of hold algorithm */
```

Figure 1. The hold model

1.2. Related empirical work

To the best of our knowledge, the earliest work addressing issues of scheduling in simulation was that of Conway *et al.*¹³ Since then, researchers have addressed the event calendar problem by proposing a host of priority queue data structures.³ Naturally, such a variety brings another problem into focus, namely that of choosing the best data structure for a given simulation application.

We believe that the earliest empirical work in response to the question of choice of event calendar algorithm is that of Vaucher and Duval. In this pioneering work, a detailed comparison of four event calendar algorithms was made. It brought the issue to the attention of other researchers and stimulated considerable debate in the area. This was the first appearance of the *hold model* (inspired by the hold operation in Simula), a methodology for comparing different event calendar implementations. In essence, the hold model attempts to assess a given algorithm by giving it inputs which are *intended* to approximate the inputs obtained by a real simulation application. The term input refers to a sequence of operations from the set $S = \{I, D\}$, where I represents *insert*, and D represents *delete*; both are operations defined on the event calendar. Evans suggests that the hold model has become widely accepted as a standard benchmark in empirical comparisons of priority queue algorithms.

Vaucher and Duval⁹ compared four calendar implementations, i.e. simple list, indexed list, and two tree-based algorithms, by initializing each with the same set of up to 200 events and executing between 4000 and 10,000 hold operations. The average time to perform a hold operation on each data structure was computed. Some generality is obtained by varying the distribution, called the *scheduling distribution*, of times used to schedule future events. This distribution dictates how event times are dispersed in the event calendar after each insertion. Vaucher and Duval used six independent scheduling distributions (three unimodal distributions: negative exponential, U[0,2], U[0.9,1.1], one bimodal distribution, one constant with value unity, and one discrete distribution D[0,1,2]). In a largely analytic follow-up study, Vaucher¹⁵ used renewal theory to obtain the asymptotic distribution of these interevent times, given the scheduling distribution. He argues that for many scheduling distributions, scanning the event calendar from the end, instead of from the beginning, improves performance.

Franta and Maly¹⁶ proposed a two-level (TL) data structure, bearing some resemblance to an indexed list of Vaucher, but where the indexed pointer points to a record in a list of secondary pointers instead of an item in the event calendar. However, their empirical work was largely restricted to comparing the performance of their algorithm to the efficient heap-based algorithms being heralded by Gonnet¹⁷ at around the same time. Whereas Gonnet's communication¹⁷ was apparently (see Reference 16) in response to the Vaucher and Duval work, and suggested heaps to be more efficient than structures proposed in Reference 9. Franta and Maly responded¹⁶ by performing experiments to show that their TL structure outperformed the heap.

The next major study was performed by Comfort,¹⁸ who worked with larger initial sizes for the event calendars than those used by Vaucher and Duval. Comfort used an initial size of up to 640 events in the event calendars, and compared seven different implementations: five list-based and two heap-based structures. Comfort also used the same scheduling distributions as used by Vaucher and Duval. Average

hold-times were measured for the different implementations using 4000 hold operations.

A more thorough empirical comparison of eight different event calendar algorithms was done by McCormack and Sargent.⁸ The first four are small variants of the linked linear list, including the simple linked list (with insertion from both ends), a multiple linked list (as used by Simscript II.5¹⁹), and a linear list with an additional pointer. The other four include the heap, the Vaucher–Duval indexed list, the Franta–Maly TL list, and a novel algorithm of Henriksen.¹⁰ Except for the heap, the others are based on linked linear lists, supplemented by special pointers for list subdivision. The latter three performed insertion from the end of the list, to reduce insertion time. Using event calendars with an initial size of 1000 events, average hold times for each implementation were obtained based on 1000 hold operations. In addition to the six scheduling distributions used in the work cited above, a hyperexponential distribution was also used.

McCormack and Sargent experimented with use of these event calendar algorithms on closed queueing systems before concluding²⁰ that no single algorithm could be recommended as being the best to use. Despite indications^{10,16} that the TL and Henriksen algorithms outperformed the Vaucher–Duval algorithms under the hold model, McCormack and Sargent report poor observed performance for the TL and Henriksen algorithms in this study. This was followed by a dissertation on event calendar implementations by McCormack,²¹ and a fine analysis of these algorithms by McCormack and Sargent.⁸ Using the analysis to support newer experiments, these authors show⁸ that the Henriksen, Vaucher–Duval, and modified heap algorithms consistently performed the best, followed by the standard heap and TL algorithm. Further, all the linked list variations exhibited poor to mediocre performance.

The next, and most thorough empirical comparison of event calendar algorithms performed to-date, was the interesting work by Jones. ¹² In this work, eleven different implementations were compared. Based on the categories in Reference 12, these include the so-called classical implementations or the linear list, heap, and (leftist) tree; also included are specialized event calendar implementations or the two-list and Henriksen's algorithm. Finally, Jones also included the more generally developed and near-optimal priority queue implementations of the binomial queue, ²² pagoda, ²³ top-down and bottom-up skew heaps, ²⁶ splay trees, ²⁵ and pairing heaps. ^{26,27}

Jones performs experiments on three different machines, using five different scheduling distributions (i.e. negative exponential, uniform, biased, bimodal, and triangular) under the hold model to make several conclusions. In each experiment, the initial size of the event calendar was fixed at 1000 events, and average hold times were measured by averaging the results of three runs of 10,000 hold operations each. The same random number seed was used in each case. Simple linked lists were found to be uniformly poor, especially for more than 50 events. Other poor performers include the leftist tree, and implicit heaps (especially for less than 20 events). Whereas pagodas, pairing heaps and skew heaps were found to perform uniformly well despite the same average-case but different worst-case behavior, binomial queues were found to exhibit erratic performance. The study concludes that Henriksen's algorithm performs very well, and is consistently outdone only by the splay tree. It should be mentioned that Jones uses a different initialization method from the one used in the Vaucher–Duval experiments, in that the event calendar is created through a sequence of random insertions and deletions: insertion probability

is chosen to be slightly larger than deletion probability, and the sequential process is made to halt when the event calendar reaches some satisfactory size for timing measurements. Also, the study is restricted to event calendars with initial (and fixed) size of 1000 events.

Though our focus is mainly on empirical work, some mention must be made of related analytic efforts. As indicated earlier, Vaucher¹⁵ made the first such effort in this direction. Engelbrecht-Wiggans and Maxwell²⁸ provide analytic results showing that time indexed (linked) lists²⁹ were superior to other synchronization mechanisms existing at that time. These efforts were followed by the detailed work of McCormack and Sargent⁸ who show that when scheduling distributions consist of mixtures, events tend to pile at the beginning of an event calendar. Reeves³⁰ analyzed the simple linear list, indexed list, and heap (two-child and three-child) implementations, arguing that though the binary and ternary heap perform well, the adaptive indexed list is the structure of choice. In a similar vein, Kingston³¹ gives a detailed analysis of the best that can be expected under the hold model for different distributions and recommends Henriksen's algorithm over the binary search tree and *p*-tree algorithms.³²

1.3. Motivation

The empirical work reported here is motivated by several factors. The only three reasonably complete empirical comparisons done in the past are those of Comfort, ¹⁸ McCormack and Sargent, ⁸ and Jones. ¹² The former two studies devote significant attention to list-based structures, whereas the latter attempts to encompass a variety of newer structures. Understandably, all empirical efforts, particularly the latter, appear to suggest that no single implementation will work best in all situations. Our first motivation came from recognizing patterns of good performance in certain structures such as the splay tree in Jones' work. ¹² This gave rise to a number of questions and piqued our curiosity.

Choosing an efficient event calendar implementation is of concern to us in our ongoing work in simulation. The area thus aware of the care that must be exercised by simulation language developers in both choosing and justifying their choice of event calendar algorithms for event processing simulations. Existing simulation systems differ in their choices; so no definitive guidelines can be obtained here. For example, GPSS and Simscript apparently used linked lists, Simula uses a *p*-tree, CSIM uses a calendar queue, and the languages SLAM and GPSS/H use Henriksen's algorithm.

In addition to general simulation-based comparisons,⁸ all previous efforts based their comparisons on the hold model. The quantity measured is the average time for a hold operation on various implementations of the event calendar in an appropriately defined steady-state operating regime. It appears that previous efforts have generally accepted the premise that the hold model approximates event calendar dynamics in real simulations. Unfortunately, the nature and quality of the approximation is not well understood. The model is typically used with a small and fixed size calendar, with independent insert and delete requests, independent interevent times, and even a single, fixed scheduling distribution. Though these assumptions may be seriously violated in simulation applications, the standard hold model has enjoyed considerable popularity as a test of event calendar performance. McCormack and Sargent⁸ introduced different but fixed calendar sizes, and mixture scheduling distributions in their

use of the hold model. Jones¹² used a clever scheme for allowing the calendar to grow to some target size before beginning measurement. However, in both cases it was assumed that events are independent, insert and delete operations strictly alternate, and calendar size remains fixed.

One major motivation behind our investigations is a relaxation of previously used assumptions. For example, in computer network or telephone-traffic problems, burst phenomena are common. This triggers *dependent*, rather than independent, sequences of insert/delete operations on an event calendar, e.g. a string of packets arriving in quick succession in simulated time causes a sequence of insertion operations. Further, this causes random fluctuations in calendar size, instead of a fixed calendar size; it also causes the scheduling of an event to depend on other events. Thus, whereas analyses used to support the (static) hold model are based on renewal theory arguments, 15,31 we argue that the (dynamic) calendar-access phenomena being modeled are more generally Markov renewal 39 and even non-Markovian.

From the experiments reported in References 8, 9, 12 and 18, it is clear that average and worst-case complexity analyses which ignore constants are not good indicators of performance. Various unknown and machine related¹² constants play an important role in dynamic behavior, possibly effecting a halving or even a doubling of simulation time. Such effects are hard to determine experimentally, and even harder to identify with theoretical tools.⁴⁰ Hence, a systematic empirical approach, along the lines of References 8, 9, 12 and 18, will permit an informative comparison. A study of dynamic behavior will also allow us to assess the efficiency of event calendar implementations under dynamic inputs and priority duplication, issues that have not been addressed previously.

In summary, past comparative assessments of such algorithms have focused mainly on hold times, smaller calendars, static tests, independent requests, and effects of independent scheduling distributions. In addition, past work appears to have focused largely on fixed vs. varying calendar sizes, pure vs. mixture scheduling distributions, and methods for initializing a calendar or simulating 'steady-state' operation of a calendar. Besides wanting to determine a class of efficient implementations for our own use, part of our motivation is to address issues of performance that have not been addressed previously, including tests with large event calendars, sensitivity to certain kinds of input sequences, dynamic performance of algorithms, effects of correlated input sequences, priority duplication, mixed distributions etc. Finally, this work includes two recently proposed implementations: the skip list⁴¹ and the calendar queue.¹¹

2. PERFORMANCE MODELS AND EMPIRICAL RESULTS

In this section we describe three kinds of models that we use for testing the performance of the different event calendar implementations. A set of empirical results is presented for each type of performance test. For clarity, the empirical tests related to each model follow immediately after the description of the model. We begin by introducing the *Markov hold model* and describe how it is used. Next, we define the notion of event calendar *efficiency* and show how this metric can be used to test how well a given event calendar implementation responds to *stress*. Next we use the Markov model in a more general manner to gauge the performance of each implementation under correlated sequences of insert and delete operations. Finally,

we present a token ring simulation application to show how such correlated inputs can arise in general simulations; it also induces effects related to duplicate priority event processing in each implementation.

As explained in Section 1.2, interaction with the event calendar in the hold model occurs through a strictly alternating sequence of insert and delete operations. An event calendar of initial size n is subject to an input sequence D_1 , I_1 , D_2 , I_2 , ..., D_k , I_k for a sufficiently large value of k. For a given calendar implementation A, let the random time to process a sequence of k such (D, I) pairs be denoted by $T_A(k|n)$. The expected time to perform a single hold operation on this n-event calendar is obtained by estimating $E[T_A(k|n)]/k$, where E[] denotes expectation. Previous empirical work^{8,9,12,18} used this estimate to compare the performance of different implementations. In generalizing the hold model to the Markov hold model, we must resort to a different estimate; instead of estimating the average time taken to perform a hold operation, we estimate the average time taken to perform a sequence of D and D operations.

2.1. Markov hold model

A natural enhancement to the hold model is one that makes calendar access requests more realistic by making access patterns more general. We do this by performing insert and delete operations on a calendar in a Markovian environment. A sequence of random variables X_1, X_2, X_3, \ldots taking values on a finite set S and satisfying the condition

$$P[X_{k+1} = i_{k+1} | X_0 = i_0, \ X_1 = i_1, \dots, X_k = i_k] = P[X_{k+1} = i_{k+1} | X_k = i_k]$$
 (1)

for all k, and all i_0 , i_1 , ..., i_k , is called a *discrete time Markov chain*.³⁹ For us $S = \{D, I\}$, and hence the Markov chain can only move between two states.

For ease of understanding, imagine a demon (the Markov chain) that can exist in only two states, called D (delete) and I (insert). Initially, assume that the demon is in state D, the *initial state* i_0 in (1). The demon is allowed to move between states in S at discrete times 0, 1, 2, ..., so that i_k represents the state the demon is in at time step k. As will be seen shortly, certain divine laws (i.e. probabilities) determine how the demon chooses between states at each time step. The defining property (1) ensures that at any time step k, only state i_k plays a direct role in determining which state the demon will be in at time k+1; states occupied prior to step k are not directly involved in this determination.

Suppose that divine law decrees that at any time step k, the demon must use a probability distribution $\{q_{i,j}(k); (i, j) \in S\}$ to determine state i_{k+1} . Then at time k, the demon must use a transition probability matrix

$$\mathbf{Q}(k) = \frac{D}{I} \begin{bmatrix} \alpha_k & 1 - \alpha_k \\ 1 - \beta_k & \beta_k \end{bmatrix}$$
 (2)

to effect transitions between the two states D and I in S; clearly $0 \le \alpha_k = q_{D,D}(k)$ ≤ 1 and $0 \le \beta_k = q_{I,I}(k) \le 1$, for all $k \ge 0$. If the demon makes a state transition

at each time step k by consulting the matrix $\mathbf{Q}(k)$, for each $k \ge 0$, then its behavior is said to be governed by a *non-homogeneous* Markov chain. If $\alpha_k = \alpha$ and $\beta_k = \beta$ for all k, then $\mathbf{Q}(k) = \mathbf{Q}$ for all k and the chain is said to be *homogeneous*.

For convenience, we restrict our attention to the homogeneous case. Practically, the Markov hold model is used as follows (see Figure 2). Given an event calendar which has been initialized with n events, the demon is initially placed in state D; the first operation on the calendar is a *delete* operation. To determine the next operation, a uniform random variate in (0, 1), say u, is generated, and its value compared to α . If $u < \alpha$, the demon remains in state D and the next operation on the calendar is also a deletion; otherwise the demon moves to state I and the next operation is an insertion operation. This procedure continues until some termination condition is met. Notice in Figure 2 that if the demon makes a transition from state D back into state D and if the event calendar is found empty, then performing a

```
begin
 • Initialize event calendar with 1000 events;
 • Delete highest priority event from calendar;
 • Synchronize clock with this event's time of occurrence;
  • state := delete; done := false; event_count := 0;
 while (not done) do {
     if (state = delete) then {
         if (calendar is empty) then print error and exit;
         Get a U(0,1) random variate u;
         if (u < \alpha) then {
          Delete highest priority event; update clock;
           state := delete; event_count := event_count + 1;
           Define new_event_time to be clock time plus
           a random variate from a scheduling distribution S_x;
          Encode new_event_time in new_event;
           Insert new_event in calendar;
           state := insert; event_count := event_count + 1;
     }else{
         Get a U(0, 1) random variate u;
         if (u < 1 - \beta) then {
           Delete highest priority event; update clock;
           state := delete; event_count := event_count + 1;
           }else{
           Define new_event_time to be clock time plus
           a random variate from a scheduling distribution S_y;
           Encode new_event_time in new_event;
           Insert new_event in calendar;
           state := insert; event_count := event_count + 1;
     } /* of if */
      if (event\_count = n) then done := true;
    } /*of while */
  end /*of Markov hold algorithm */
```

Figure 2. Markov hold model

Relation

Generality

Comparison	Standard Hold	Markov Hold	
Input sequence	D, I, D, I, D, I	random, function of α and β	
Calendar size	fixed	varies (grows, shrinks)	
Scheduling distribution	fixed	allows mixture distributions at transisions: $D \rightarrow I$, $I \rightarrow I$	
Behavior	static	dynamic	

independent/dependent

correlated/uncorrelated

general

independent,

uncorrelated

special case of Markov

hold model with $\alpha = \beta = 0$

Table I. Comparison: standard and Markov hold models

delete operation on the calendar will result in an error. We ensure that this does not happen in our experiments by choosing an initial event calendar size and number of operations performed on the calendar with some care. An alternative strategy is to bias the Markov chain by moving the demon to state *I* whenever it is in state *D* and the calendar is empty. However, this makes the two-state process non-Markovian and is not recommended; besides effecting results, it introduces the possibility of an indefinite length zero—one oscillation in calendar size.

2.2. Random input tests

Our first set of tests is based largely on subjecting each event calendar implementation to a sequence of operations in which insertion and deletion requests occur randomly. The simplest sequence of randomly generated requests is one in which insertion requests and deletion requests occur *independently* with fixed probabilities P(I) and P(D), respectively. This independence is obtained by making the two rows of the transition probability matrix \mathbf{Q} identical, so that $\alpha = 1 - \beta$. This will yield $P(D) = \alpha$ and $P(I) = \beta$.

As can be seen in Table I, the Markov hold model is a generalization of the standard hold model; when $\alpha=\beta=0$, the Markov hold model reduces to the standard hold model. We identify five types of potentially interesting sequences for input to each event calendar implementation. These are summarized in Table II. Each

	<u> </u>	
Type	Description	Request are
1	Favor neither Favor deletion	independent independent
3	Favor insertion	independent
4	Standard hold	independent
5	Positive correlation	dependent

Table II. Random input sequence types

type of input induces a different kind of event calendar behavior, and thus offers a wide range of performance tests. Type 1 makes insertion and deletion requests randomly, without bias toward either kind of operation; type 2 favors deletion operations, whereas type 3 favors insertion operations; type 4 is simply the standard hold model. Type 5 triggers positively correlated sequences of insertions and deletions in that like operations tend to occur consecutively with high probability. Of the five kinds of inputs described, this is the only kind in which insertion and deletion requests are *dependent*.

Empirical results for random input tests

We perform five different tests by subjecting all calendar implementations to the five types of inputs shown in Table II. The quantity measured in each case is the average completion time for a total of N operations, with the event calendar initialized to hold 1000 events. This is accomplished using the Jones initialization scheme: insertions are done with probability 0.6, so that the calendar grows slowly to the desired initial size. Observe that the measure described differs from that used in past work. Instead of working with an event calendar of *fixed size* and computing average hold time by repeating the hold (i.e. insert and delete) operation some given number of times, we measure the amount of time required to perform N consecutive operations. This is closer to what happens in a real simulation in that the size of the event calendar changes dynamically, and insert and delete operations do not strictly alternate. More importantly, operation times are a function of the changing size of the calendar, and this is accounted for in our measurement.

The sequence of insert and delete requests is Markov renewal in that the embedded operation sequence is Markovian, and the scheduling distributions are non-exponential. On a transition from an insertion operation to another insertion operation, the scheduling distribution used for the event corresponding to the latter insertion is two-component hyperexponential with a mean of 2 and a standard deviation of 4. On a transition from a deletion operation to an insertion operation the scheduling distribution used is an Erlangian with a mean of 2 and a standard deviation of 1.

In Figure 3 can be seen a graph of completion times for each of the event calendar implementations under inputs satisfying P(D) = P(I), i.e. inputs of Type 1. If the scheduled interevent times are purely exponential, then the left to right ordering of the calendar implementations on the horizontal axis in this Figure corresponds to the order of improving performance; the linked list and two-list exhibit worst performance, and the splay tree and calendar queue exhibit the best performance. However, owing to space considerations, and the fact that the interevent time distributions tend to possess coefficients of variation greater than unity, the measurements in all the Figures shown here are based on mixture distributions. It is interesting to observe that the performance of the leftist tree deteriorates, although otherwise the overall trend remains roughly the same; the only other exception is the calendar queue, whose performance also appears to fall below the performance of the splay tree for large N. The circle on the curve for N = 100,000 indicates that Henriksen's algorithm could not be tested at this value of N due to memory limitations related to our implementation.

The performance of the different implementations under inputs of Type 2 (favoring delete operations) can be seen in Figure 4. The completion times are now much

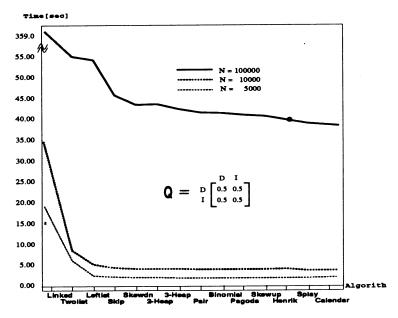


Figure 3. Type 1 with Erlang-k and H₂

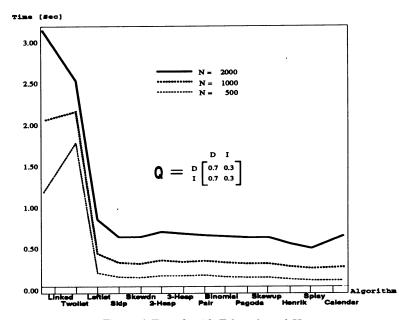


Figure 4. Type 2 with Erlang-k and H₂

smaller, since the algorithms tend to work with smaller calendars. Once again, the general trend appears to be the same, but the splay tree outperforms the calendar queue for large N. At this point, the data suggests that the splay tree tends to have the best overall performance, for the given tests. However, consider the empirical results shown in Figure 5 for the Type 3 inputs (favoring insertion operations). Although the overall trend appears roughly the same, the performance of the splay tree deteriorates considerably, whereas the performance of the calendar queue is improved. This suggests that the splay tree is overly sensitive to repeated insertion operations, whereas the calendar queue is overly sensitive to repeated deletion operations.

The results in Figure 6 correspond to Type 4 input sequences, which are the same as those given by the hold model. Here the performance of the leftist tree is poorer, and once again the calendar queue is outdone by the splay tree for large inputs. The overall performance trend appears to be in some agreement with the results of Jones. Finally, the results in Figure 7 show completion times for an input sequence consisting of strings of consecutive delete operations and strings of consecutive insert operations. Here also the overall trend appears to be the same, with the splay tree outperforming the calendar queue for large inputs.

The results of the five kinds of tests shown above appear to indicate that the splay tree and the calendar queue are excellent performers under random, unbiased inputs. However the splay responds poorly to long sequences of insertion operations, whereas the calendar queue responds poorly whenever consecutive operations entail repeated resizing. Contrary to data presented in Reference 11, our experiences suggest that the calendar queue does not appear to clearly outdo the splay tree. This is seen when the calendar queue is subjected to demanding input sequences which trigger expensive resizing operations.⁴²

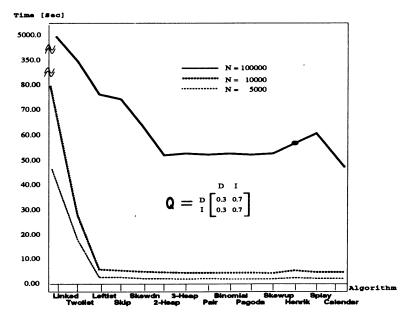


Figure 5. Type 3 with Erlang-k and H₂

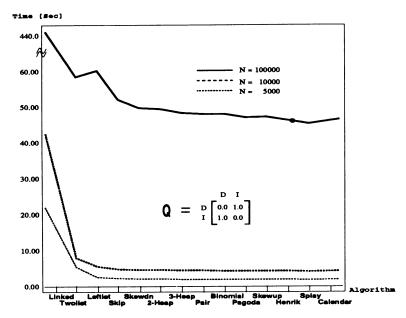


Figure 6. Type 4 with Erlang-k and H₂

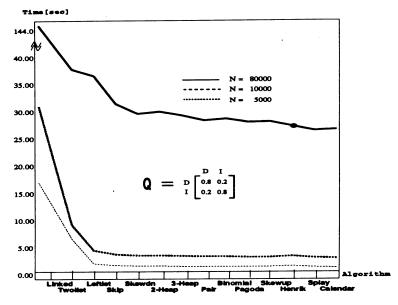


Figure 7. Type 5 with Erlang-k and H₂

2.3. Drift-based stress tests

Previous comparative evaluations^{8,9,12,18} of event calendars appear to have focused largely on the effects of fixed calendar sizes and varying scheduling distributions. Influenced by experiences in telephone-traffic and network simulations, we are aware

that event calendars can grow and shrink in an unpredictable manner, making effects of varying scheduling distributions small in comparison to swings in calendar sizes. Swings toward larger sized calendars can have a particularly detrimental effect on performance because of machine-related overheads that can be induced, such as cache and memory contention. Similarly, our experiments will show that swings towards smaller sized calendars can also have a strange effect on performance. Thus, the dynamic performance of event calendars eludes static tests such as those induced by the standard hold model.

An event calendar implementation is said to be *efficient* if its performance is not adversely affected by sudden calendar growth or shrinkage. If one thinks of a sequence of I and D operations as an input to the calendar, and the time taken for it to perform on this sequence as its output, then the output should not be overly sensitive to long sequences of insertion operations or deletion operations. Whereas the Markov hold model can be used to test such dynamic effects through random growths (high insertion probabilities) and random shrinkages (high deletion probabilities), it does not given an indication of how robust a calendar implementation is under strong growth and strong shrinkage. To handle this, we introduce two special kinds of tests based on the notion of drifts⁴³ in Markov chains.

For any event calendar implementation A, let $I_A(k|n)$ denote the time required to perform k event insertions, given that the calendar size is initially n. We define

$$D_A^+(k|n) = E[I_A(2k|n)] - E[I_A(k|n)]$$
(3)

to be the *k-insertion drift* of implementation A, for n and k as positive integers. With k fixed, the quantity $D_A^+(k|n)$ is a measure of how well implementation A handles repeated insertions as a function of size. It should be noted that (3) makes use of linearity of the expectation operator under the condition that the initial calendar size is n; in this respect, it satisfies the definition of average drift.⁴³ To eliminate spurious effects, we normalize the drift in (3) to obtain the *growth rate* of implementation A as

$$G_A(k|n) = \frac{D_A^+(k|n)}{E[I_A(k|n)]}$$
(4)

$$=g_A(k,n)-1\tag{5}$$

where

$$g_A(k, n) = \frac{E[I_A(2k|n)]}{E[I_A(k|n)]}$$
(6)

is the implementation's growth efficiency.* For example, $g_A(n/2, n)$ describes how stably implementation A doubles its size, assuming a string of insertion operations. If an implementation remains relatively unaffected by n, and on the average does

^{*} As seen in Table III, the calendar queue's behavior depends on how much resizing occurs and in which phase. Efficiency may appear high if resizing does not occur in either phase.

roughtly the same amount of work for each insertion operation, then for this implementation $g_A(n/2, n) \approx 2$, so that $G_A(n/2|n) \approx 1$ and growth efficiency is linear. Similarly, for a given event calendar implementation A, define $D_A(k|n)$ to be the time required to perform k event deletions, given that the size is initially n. We define

$$D_{A}^{-}(k|n) = E[D_{A}(2k|n)] - E[D_{A}(k|n)]$$
(7)

to be the *k*-deletion drift of implementation A. The quantity $D_A^-(k|n)$ is a measure of how well implementation A handles repeated deletions as a function of size. Normalizing the average drift in (3) yields the *shrink rate* of implementation A as

$$S_A(k|n) = \frac{D_A^-(k|n)}{E[D_A(k|n)]}$$
 (8)

$$= s_A(k, n) - 1 \tag{9}$$

where

$$s_A(k, n) = \frac{E[D_A(2k|n)]}{E[D_A(k|n)]}$$
(10)

is the implementation's *shrink efficiency*.* For example, $s_A(n/2, n)$ describes how stably implementation A halves its size, assuming a string of deletion operations. As before, an algorithm which does roughly the same amount of work for each deletion operation and is relatively unaffected by n will satisfy $s_A(n/2, n) \approx 2$, so that $S_A(n/2|n) \approx 1$ and shrinkage efficiency is linear.

Table III. Growth times (μ-se	s) and efficiency:	n = 8192.	k =	n/2
-------------------------------	--------------------	-----------	-----	-----

Type	$E[I_A(k n)]$	$E[I_A(2k n)]$	$g_A(k, n)$
Calendar	240	252	1.045
Splay	427	902	2.112
Henrik	252	505	2.003
Skewup	264	514	1.946
Pagoda	295	530	1.796
Binomial	282	574	2.035
Pairing	277	539	1.945
3-Heap	261	534	2.045
2-Heap	273	529	1.937
Skewdn	351	720	2.051
Skip	510	1033	2.025
Leftist	535	1187	2.218
Twolist	11,244	21,493	1.911
Link	96,687	216,328	2.260

^{*} As seen in Table IV, the calendar queue's efficiency can be poor if resizing occurs in phase 2 but not in phase 1.

1.964

Type	$E[D_A(k n)]$	$E[D_A(2k n)]$	$s_A(k, n)$
			<i>5A</i> (<i>i</i> , <i>i</i>)
Calendar	53	860	16.301
Splay	38	76	2.021
Henrik	73	136	1.867
Skewup	248	460	1.854
Pagoda	295	407	2.099
Binomial	282	530	1.879
Pairing	257	479	1.863
3-Heap	302	554	1.834
2-Heap	313	579	1.849
Skewdn	99	182	1.838
Skip	131	255	1.946
Leftist	83	162	1.951
Twolist	6960	10,372	1.490

28

Table IV. Shrink times (μ -secs) and efficiency: n = 8192, k = n/2

Empirical results for stress tests

Link

The quantities $G_A(\cdot|\cdot)$ and $S_A(\cdot|\cdot)$ can be used to measure how well (i.e. stably) an event calendar implementation performs under extreme situations as a function of its current size. In our experiments, we study these quantities via the growth efficiency $g_A(n/2, n)$ (i.e. Markov hold model with $\alpha = 0$, $\beta = 1$) and shrink efficiency $s_A(n/2, n)$ (Markov hold model with $\alpha = 1$, $\beta = 0$), respectively. Since these quantities measure ratios of performance, they are less sensitive to machine effects; they relate each implementation's performance under stress to its own unstressed performance. This gives an analyst one approach to defining a practical notion of efficiency under growth and shrinkage.

The quantity $g_A(n/2, n)$ is computed by first initializing the calendar to hold n = 1 2^{j} events, using the gradual insertion scheme described in the previous set of tests. We next estimate the completion time of the first phase (i.e. phase 1) comprising a sequence of n/2 insertion operations. This is followed by estimation of the second phase (i.e. phase 2) comprising a sequence of another n/2 insertion operations, though the calendar size at the start of the second phase is n + n/2. The average completion time for the two phases is estimated by repeating the experiment thirty times, each time using a different starting seed, for each value of j, $6 \le j \le 13$. An identical procedure is applied in obtaining $s_A(n/2)$, except that both phases now consist of deletion operations instead of insertion operations. These phase completion times are used to obtain the ratios $g_A(n/2, n)$ and $s_A(n/2, n)$ as functions of n. Each ratio is given by the sum of the times for both phases divided by the time for the first phase. To gauge the performance of each simulation calendar implementation under this test, we determine the range (i.e. minimum to maximum time required to process an input string) of $g_A(n/2, n)$ and $s_A(n/2, n)$ over all n for each implementation. This range gives an indication of how much these efficiency metrics can vary for each implementation.

In Figure 8 is shown the range of values of $g_A(n/2, n)$ given by each event calendar implementation. As expected, nearly all implementations show a growth

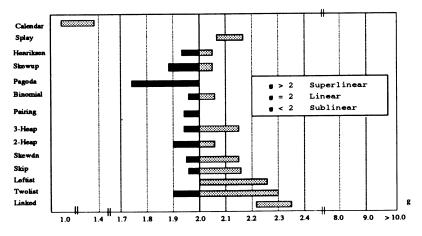


Figure 8. Growth efficiency

efficiency that appears to be centered roughly around the number 2. However, some implementations exhibit positive skews; examples include the twolist, leftist, skip list, skew-down heap, (implicit) three-heap, splay, and in particular the linked list. Hence, taking growth efficiency as a measure of stability, these implementations exhibit a tendency to require more work per insertion with increasing event calendar sizes. The binomial heap and the algorithm of Henriksen appear to do well in growth efficiency, while the pairing heap and the pagoda structures seem to show an interesting tendency to become more efficient under growth with increasing calendar sizes. Calendar queue performance is sensitive to the amount of resizing that occurs and when. For example, high growth efficiency results if resizing occurs in phase 1 but not in phase 2 (see Figure 8). Conversely, efficiency will appear poor if resizing occurs in phase 2 but not in phase 1 (see Figure 9).

In Figure 9 can be seen a similar graph for the range of shrink efficiency values

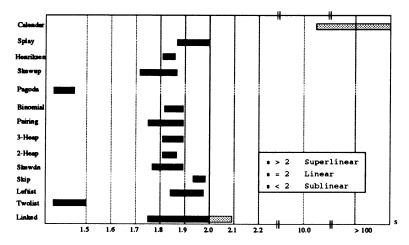


Figure 9. Shrinkage efficiency

 $s_A(n/2, n)$ obtained by each implementation. Interestingly, except for the calendar queue, all implementations exhibit negative skews, suggesting that smaller sized event calendars are more stable. The performance of the splay, skip list and leftist tree structures is fair, and excellent performers include the Pagoda and the twolist. The poor shrinkage efficiency of the calendar queue is attributed to resizing.⁴²

For completeness, we also include a test where each structure is allowed randomly to grow and shrink with equal probability. That is, each implementation is subjected to a sequence of operations in which insertion requests occur with the same probability as deletion requests; and each request is *independent* of the previous request. The results of this test can be seen in Figure 10. Under this test, which we call the random efficiency model (and hence denote by r), the calendar queue and pagoda perform very well, and the other implementations show a reasonable performance. The splay and leftist tree structures exhibit lower growth efficiency than the others, with the two-list showing a large range and overall worst performance. The growth, shrinkage and random efficiency tests appears to suggest that the pagoda structure is very stable in its ability to respond to strings of insertion, deletion, or even random requests; under growth it exhibits a large efficiency range, while otherwise its efficiency range is small. Small efficiency ranges are suggestive of 'stability', in that the ratios do not oscillate wildly such as in the cases of the calendar queue, linked list, and twolist structures. Consistently good performers in the sense of this stability include the algorithm of Henriksen, the pairing heap, and the two-heap. To differentiate this kind of stability from the notion of time-stability, we label such implementations as *structure-stable*.

The apparent poor performance of the calendar queue is a function of test parameters. Since the calendar queue is tailored to undergo resize operations under certain conditions, its performance under such efficiency tests may not be accurately obtained if these parameters are ignored.^{11,42} Growth or shrinkage efficiency will be poor if time spent in performing the resize operation enlarges the numerators in g_A and s_A , respectively. On the other hand, growth or shrinkage efficiency will be good if this time enlarges the denominators in g_A and s_A , respectively. Thus, the calendar queue's efficiency is sensitive to its input data.

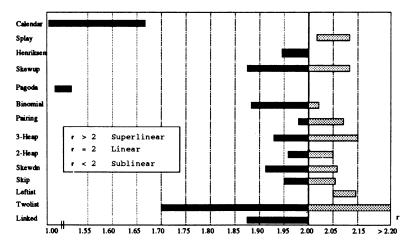


Figure 10. Random efficiency

2.4. Markovian input tests

As explained in Section 2.1, the Markov hold model can be used to generate dependent sequences of I and D operations. Since the standard hold model only tests calendar performance on hold operations statically, given a fixed initial calendar size, it ignores the manner in which different event calendar implementations respond to more general sequences of insert and delete operations. A given event calendar implementation A may perform better than another calendar implementation B for an input sequence of one kind, and worse than B for an input sequence of another kind. Since such performance depends largely on the dynamics of calendar behavior as a function of input, the Markov hold model can be used to parametrize input sequences. We accomplish this through use of the *correlation coefficient* ρ .

For the two-state Markov chain $\{X_n\}$ introduced at the beginning of this section, it can be shown that

$$\rho = \alpha + \beta - 1 \tag{11}$$

so that $-1 \le \rho \le 1$. An input sequence is negatively correlated if $\rho < 0$. For such inputs consecutive operations tend to be different; as an extreme example, $\rho = -1$ corresponds to the *hold model*, where an input sequence is made up of strictly alternating I and D operations. When $\rho = 0$, consecutive operations are independent. When $\rho > 0$, an input sequence is positively correlated; an extreme example is the situation $\rho = 1$ where the input is made up of an infinite sequence of either insert or delete operations. For positively correlated inputs, consecutive operations tend to be alike.

Using the initialization scheme described earlier and an initial size of 5000 events, each calendar implementation is subjected to an input sequence of 5000 operations for $-0.99 \le \rho \le 0.99$; the maximum and minimum completion times for each implementation are recorded. An average completion time is estimated based on 30 independent runs of 5000 operations for each implementation, and this is done for each value of ρ . A different pair of random number seeds was used in each run, where one seed was used to generate the stream of initial events, and the other seed was used to generate the input sequence. Two cases must be distinguished: one in which $\alpha \ne \beta$, and the other in which $\alpha = \beta$. In the former case insertion and deletion operations occur with unequal probabilities in an input sequence, whereas these probabilities are equal in the latter case.

Empirical results for Markovian inputs

We measure the completion time range (i.e. minimum to maximum time required to process an input string) for each calendar implementation as a function of ρ . First, consider the case in which $\alpha = \beta$. When insertion and deletion probabilities are equal, correlation does not appear to have significant effect on completion time ranges, except for the two-list and linked list structures and to a lesser extent, Henriksen's algorithm. The results of this experiment can be seen in Figure 11. On the other hand, when insertion and deletion probabilities are unequal (which is the case when $\alpha \neq \beta$), the different calendar implementations appear to respond differently to correlated inputs. The results of this experiment can be seen in Figure 12. Calendar implementations with particularly small completion time ranges

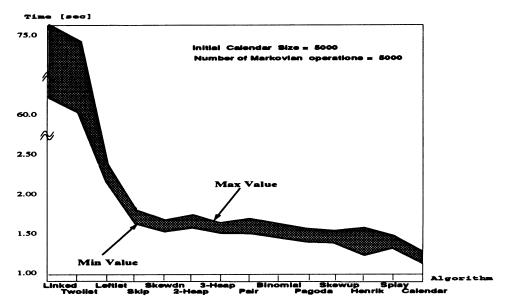


Figure 11. Correlation effects ($\alpha = \beta$)

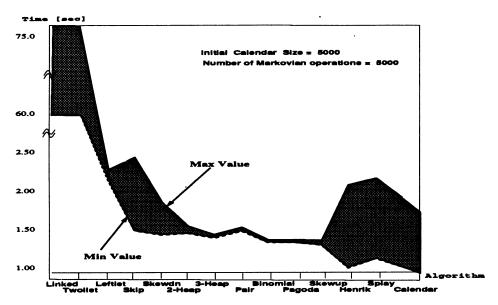


Figure 12. Correlation effects ($\alpha \neq \beta$)

include the leftist tree, implicit heaps, pairing heap, binomial heap, pagoda and skew-down heap structures. Remarkably, Henriksen's algorithm, the splay tree, and the calendar queue show wide completion time ranges. Like the less efficient skip list, two list, and linked list, these structures appear to show some sensitivity to Markovian inputs.

Since certain event calendar implementations show good overall performance under

the random input model, investigating their performance more closely under correlated inputs when $\alpha \neq \beta$ is worth the extra effort. In Figure 13 can be seen a performance comparison of the calendar queue and two-heap implementations. The shaded area represents the completion time range for the two-heap structure, whereas the space between the upper and lower dotted lines represents a completion time range for the calendar queue. This poor performance on the part of the calendar queue can be attributed to its periodic resizing requirement. In a similar fashion, a performance comparison of Henriksen's algorithm, the splay tree, and the pagoda structures can be seen in Figure 14. Here the innermost shaded area represents the completion time range for the efficient pagoda structure; the space between the dotted lines shows the range for the splay tree. It is worth noting that the more specialized structures (e.g. Henriksen's algorithm, calendar queue and splay tree) appear to exhibit poor performance in the sense of wide completion time ranges under Markovian inputs.

2.5. Token ring simulation

In addition to the Markov hold model and efficiency tests, we also include a comparison of event calendars based on a simulation of a token ring⁴⁴ network protocol. The protocol can be modeled as a multiqueue system with a cyclic server,⁴⁵ where the multiqueue system is represented by *N* independent computer stations situated on a ring. Messages made up of packets are generated by each station for transmission to other stations on the ring. A single token is passed unidirectionally from one station to its successor on the ring, to provide stations with a mechanism for conflict-free access to the ring for packet transmissions. A station which acquires the token and has queued packets is allowed to complete transmission of a single packet before relinquishing control of the token to the

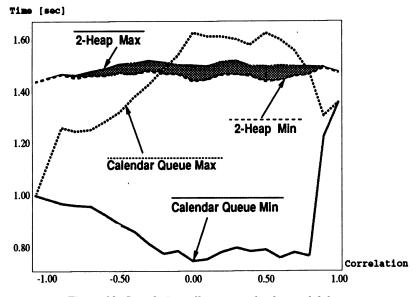


Figure 13. Correlation effects on calendar and 2-heap

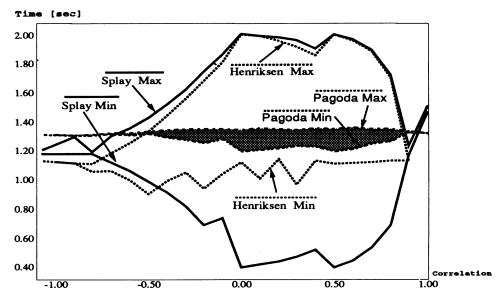


Figure 14. Correlation effects on Henriksen, Splay and Pagoda

succeeding station on the ring. It is of some interest to determine queueing characteristics of packets at different stations as a function of ring parameters and station traffic.

The parameters of the model include message interarrival time distributions, and packet transmission time distributions at the different stations on the ring. For convenience, we assume that all interarrival time distributions are identical. Also, for convenience, assume that all packet transmission time distributions are identical. Finally, assume that the token-passing time between consecutive stations on the ring is a small constant, a function of ring delay.

Since packet traffic generated at a given station is rarely (if ever) made up of independent packet units, a more realistic model of traffic will require correlated packet input. Such a dependent stream of packet arrivals can be generated by resorting to arrivals in a random environment. We can do this by assuming a Poisson-type input with an intensity parameter that changes in Markov fashion. For example, on the average the message interarrival times can be either small or large. As explained in the beginning of this section, assume there exists a demon (Markov chain) that lives in one of two states: *low* and *high*. When in the high state, the demon sets the Poisson parameter to λ_{low} . Thus the demon's behavior (and hence the message traffic generated at stations) can be controlled through probabilities α and β . In addition to Markovian message input, we assume that a message corresponds to a random number of packets.

Empirical results for token ring simulation

A thirty-station token ring model was simulated using hyperexponential packet transmission times and uniformly distributed message sizes (in integral packet units)

ranging from 1 to 500. The token-passing time between consecutive stations on the ring was set to a small constant. The model was run for a total of 25,000 events and the sequence of insertion and deletion operations saved for input to the fourteen event calendar algorithms. Next, timings were obtained by subjecting each calendar algorithm to this input sequence.

The Markov hold model was used to independently generate a sequence of insert and delete operations which mimicked the sequence given by the token ring simulation. This was done by choosing the α and β probabilities in the transition appropriately: because the token ring simulation made positively correlated requests to the event calendar, the Markov hold model was made to generate a positively correlated sequence of insert and delete operations. Each insert and delete request was expanded into a string of K identical operations, where K was a uniformly distributed integer between 1 and 500. In this way, the Markov hold model was used to measure the amount of time each algorithm would require to process 25,000 such events, resembling the token ring model's sequence of insert and delete operations.

The two sets of timings described above were obtained by executing all fourteen algorithms on a Sequent Symmetry. The results can be seen in Table V. The first column of numbers shows the amount of time each algorithm required to process the insertion/deletion requests generated by the token ring simulation. The second column shows the time taken by each algorithm on the data generated by the Markov hold model. The algorithms are ranked in decreasing order of execution time required to process the actual simulation data (i.e. decreasing order of times with respect to the first column).

The presence of batch arrivals in the simulation causes a previously unexamined kind of input sequence to the event calendar algorithms. A large number of packets arriving simultaneously requires the algorithm to insert this number of *equal priority* packets into the event calendar. The existence of large numbers of such *duplicate*

Table V. Algorithm ranking: simulation vs. Markov hold model

Algorithm	Token ring simulation	Markov hold model
Calendar (standard)	94.69	57.19
Linked list	48.88	38.17
Twolist	46.02	33.46
Pagoda	26.51	19.86
Skip	15.14	16.10
Calendar (optimized)	12.28	15.11
3-Heap	12.02	15.83
2-Heap	11.84	15.63
Binomial	11.79	15.60
Henriksen	10.67	14.72
Skewup	10.35	14.51
Skewdn	10.28	14.46
Leftist	10.27	14.22
Pairing	10.21	14.13
Splay	10.01	13.89

events can evoke poor behavior in some algorithms. Consider, for example, the calendar queue algorithm. Once a destination bucket has been located, the algorithm traverses a sorted linked list in the bucket in order to insert the event in an appropriate location. Given that a bucket contains several duplicate events, the algorithm takes a significant amount of time to insert a new duplicate into this bucket because of the list traversal. This explains the standard calendar queue's (labeled 'standard') large timing requirements in Table V. If the insertion portion of the algorithm is modified so that a new duplicate event is inserted at the head of the queue, instead of the tail, then timings improve dramatically. This can be seen in the timing shown for a version of the calendar queue (labeled 'optimized') designed to do this. This optimization causes the resulting implementation to be *time-unstable*, since these duplicate events will be deleted from the queue in the reverse order.

From Table V, it can be seen that the Markov hold model yields a fairly accurate picture of the relative ranking of the times required by the different event calendar algorithms for the token ring application. Except for the optimized version of the calendar queue, which the Markov hold model incorrectly predicts will be slightly better than the binomial queue, the relative rankings predicted are the same as those given by the simulation data. Although this does not imply that the model will work as well in all situations, it does suggest two points. First, if the Markov hold parameters can be chosen such that the model mimics the simulation application's calendar requests well, then the predicted ranking will be good. Secondly, the model is consistent in demonstrating good performance by some algorithms, and poor performance by others.

3. CONCLUSIONS

Having attempted to obtain a comparative ranking of several different event calendar algorithms, we find that we are unable to recommend any one particular algorithm as being the best to use in all situations. This conclusion is in accord with the conclusions outlined by two previous empirical studies. However, our approach to arriving at this conclusion has given us some insight into methods for making choices. Although such methods may use the Markov hold model to narrow down the choice to a few algorithms, they must involve an interaction between the executing simulation application and the candidate algorithm. We have found that event calendar algorithms can behave differently during simulation execution due to paging and machine-dependent effects. A detailed study of such effects is the subject of our ongoing work, with results to be presented in a sequel to this paper.

Though we hesitate to recommend any one particular algorithm as the event calendar implementation of choice, our experience suggests the heap to be a fairly structure-stable algorithm. Its execution time performance however is outdone by the splay tree and the calendar queue. The splay tree is consistently structure-stable, except during phases of heavy insertion. We found the calendar queue to exhibit excellent performance at times, but with a tendency for erratic behavior.

Our current work seeks to carry the event calendar algorithm selection process further by providing an event calendar library and interface which allows the user a choice of several calendar algorithms. A practically useful selection strategy would execute the application using a set of specified calendar algorithms for a pilot run.

The results of this run could then be used as a fairly reliable indicator of a good event calendar algorithm for a production run.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the help of Doug Jones in this research effort. Thanks are also due to Jeff Kingston for making his early work available to us, and to Dale Rahn for his help with the Calendar queue experiments.

This research was supported in part by NSF CCR-9102331, NATO CRG-900108, ONR-9310233 and ARO-93G0045.

APPENDIX: EVENT CALENDAR IMPLEMENTATIONS

In this appendix we briefly describe the different simulation event calendar implementations that are compared. For convenience, we separate these implementations into four classes: list structures, balanced tree-structures, unbalanced tree-structures, and self-adjusting structures.

List implementations

As variants of the oldest and simplest linear list structure, the list structures are really classical implementations of event calendars. Initially, these were implemented using arrays. With the advent of pointers, list structures are now often implemented with more sophisticated pointer-based algorithms.

Linear list

We use a simple sorted linked list to implement this structure. Because it is sorted, a deletion operation can be done in O(1) time. An insertion operation requires searching for the correct insertion location, and thus results in an average and worst-case complexity of O(n).

Two-list

The two-list⁴⁶ is an early adaptation of the simple linked list. It splits the set of calendar elements into two sets; one which is short and sorted, and another which is long and unsorted. The split point coincides with an instant in future (simulation) time. For deletion, an attempt is made to retrieve the first event from the short list. If this list is found empty, a new split point in future time is determined, and the short and long lists redefined. The idea is to choose a split point such that the length of the short list is \sqrt{n} on the average. Owing to set redefinition, deletion complexity can be as bad as O(n). However, O(1) deletions from the short list occur most frequently for well-chosen split points, and the average deletion complexity is claimed to be $O(\sqrt{n})$. Because of the length of the short set, average insertion complexity is $O(\sqrt{n})$.

Henriksen's algorithm

The Henriksen algorithm¹⁰ is a modification of the Vaucher–Duval⁹ index list algorithm where a vectored binary search of the index set is used to locate a search

initiation point. The algorithm efficiently combines list insertion with binary search. With a doubly-linked structure for the event list and the supplementary search vector whose entries are pairs of event-times and pointers for corresponding event-list entries, the algorithm accomplishes a deletion operation in O(1) time. Insertion is more complicated, requiring a binary search of the vector to locate the first entry with scheduled time exceeding the to-be inserted event's time, and then performing a linear search to the left, along an associated sublist. Kingston³² derives an $O(\sqrt{n})$ upper bound for event insertion. This algorithm is at present used in the GPSS/H and SLAM simulation languages.³

Skip list

Skip lists⁴¹ have been proposed as (probabilistic) alternatives to balanced tree-structures. A probabilistic balance is achieved through use of random numbers during list construction, in contrast to the deterministic balancing schemes of tree-structures. Skip lists are constructed with linked lists and supplementary pointers that are used to skip over intermediate events in the list. Each event in the list is stored at a randomly chosen level during its insertion, and this level is independent of the current size of the list. Insertion and deletion is performed by scanning the events through list levels in search mode, followed by a splicing operation. Event insertion can increase, and event deletion can decrease the number of current levels in a list. In Reference 41 it is claimed that the cost of insertion/deletion is dominated by the cost of search, which is shown to be $O(\log n)$ for a list with n events.

Unbalanced tree implementations

Tree-based structures are popular because of the potential for a logarithmic relationship between tree-height and the complexity of insert and delete operations. Though this necessarily entails strict balance, not all tree structures emphasize balance.

Implicit d-heap

It is suggested³ that the original 2-heap structure⁴⁷ was invented for event calendar operations. It was extended by Johnson⁴⁸ to d-heaps, for d > 2. A d-heap is a tree-structure, characterized by a simple property called the heap property. The property is said to hold if the priority (event-time) of any node (event) is higher than (has a scheduled occurrence time at least as early as) the priority (event-time) of each of its children.

A deletion operation begins with root extraction; this is the earliest event. Since this operation breaks the structure apart, some post-processing is required for heap maintenance. The rightmost leaf in the lowest level of the tree is temporarily moved into the root position. Next, a percolation process is initiated during which the root node is compared with each of its children; a swap in position occurs, if necessary, for heap maintenance. In this way, the displaced event percolates to an appropriate position in the tree while some other event moves up to occupy the root node and thus restore the heap structure. Insertion is accomplished by placing the event at the base of the heap (i.e. in the rightmost leaf at lowest level) and repeating the latter portion of the deletion algorithm. Both insertion and deletion have an average cost

complexity of $O(\log n)$. Heaps are easily implemented with arrays, in which case they are called *implicit* heaps.

Pagoda

The pagoda structure was invented by Francon, Viennot and Vuillemin²³ as a mergeable priority queue structure. It is based on the heap-ordered binary tree and similar to the leftist tree. The pagoda does not attempt to maintain a balanced structure. The average cost of an insert or delete operation on a pagoda is $O(\log n)$. In an extreme, it is possible for a pagoda to degenerate into a sorted list.

Balanced-tree implementations

The tendency of unbalanced tree-structures such as the pagoda to degenerate into linear list structures and effect costly insert and delete operations stimulated a search for more balanced structures.

Leftist-heap

The leftist heap was proposed by Crane⁴⁹ as an alternative to the *d*-heap. Unlike the heap which is non-mergeable, the leftist heap is mergeable. If x is a node in a full binary tree, define its $rank \ r(x)$ to be the minimum length of a path from x to an external node. For example, r(x) = 0 if x is an external node, and $r(x) = 1 + \min(r(\text{left}(x)), \ r(\text{right}(x)))$ if x is an internal node. A leftist tree is heap-ordered full binary tree satisfying

$$r(\operatorname{left}(x)) \ge r(\operatorname{right}(x)), \quad \forall x$$

where x is an internal node. The right path of a leftist tree is a shortest path from the root to an external node, and at most $O(\log n)$ in length. A leftist heap is a heap-ordered leftist tree containing one item per internal node.⁵⁰

For insertion into a leftist heap, an event is viewed as a 1-node heap and merged with the existing heap. Deletion of an event is accomplished by extracting the root and merging the left and right subtrees. Since the leftist heap merge can be done in $O(\log n)$ time, both insertions and deletions can be done in $O(\log n)$ time.

Binomial heap

A binomial heap T^{22} is a collection of binomial trees. It is a mergeable priority queue similar to the leftist heap and pagoda structures; it is considerably faster than the leftist tree. A binomial tree is an ordered tree defined recursively, with the following binomial heap properties:⁵¹

- 1. Each binomial tree T is *heap-ordered*. Thus the root contains the highest priority value.
- 2. There is at most one binomial tree in T with a root of a given degree. Thus if the binomial heap T has n nodes, then it consists of at most $\lfloor \log n \rfloor + 1$ binomial trees.

Since there are at most $\lfloor \log n \rfloor + 1$ binomial trees, there are at most $\lfloor \log n \rfloor + 1$

1 roots that need to be examined. A minimum value is found in time proportional to $\lfloor \log n \rfloor$. Insertion time is O(1) for a one-node binomial heap, and $O(\log n)$ for an n-node binomial heap.

Self-adjusting implementations

Both tree and list based structures can be shown to exhibit high access costs due to a tendency to degenerate into poor shapes. To counter such effects, a variety of periodic shape readjustment schemes have been proposed for both structures.

Splay tree

The splay tree⁵¹ was developed by Sleator and Tarjan²⁵ and is another variation of the balanced binary search tree. It uses a tree-restructuring technique called *splaying*, essentially a sequence of tree rotations that help move a node up towards the root. In the standard balanced tree, balancing is done either through rotations (e.g. AVL trees) or through manipulation of node degree in the tree (e.g. 2–3 trees).³ Consequently, after each insertion or deletion, some time must be expended to maintain tree balance. The splay tree tends to adjust itself to access requests by using rotations to move nodes on highly accessed branches closer to the root; this makes them more readily accessible. Splay trees maintain balance without explicit conditions. Instead of affixing balance information to each node, balancing is performed implicitly on each tree access. The amortized cost²⁵ of insert and delete operations on an *n*-node splay is $O(\log n)$.

Skew heap

The skew heap structure is a self-adjusting heap and is related to the leftist heap. The idea is to adjust the structure in a simple manner with each access, so that future access efficiency is improved. Central to all skew heap operations is the *melding* operation, which merges two heaps into one. One version of the skew heap (the skew down) uses bottom-up melding and has been likened to the pagoda structure in operation; the other version (the skew up) uses top-down melding and has been likened to the leftist tree structure in operation. Although the worst-case cost of an insert or delete is O(n) for an n-node skew heap, the amortized cost is known to be bounded from above by $O(\log n)$.

Pairing heap

The pairing heap²⁶ is another self-adjusting heap structure, based on the binomial heap. The central operation here is a *linking* operation, which combines heap-ordered trees. Since this is an O(1) cost operation, and since an insert operation is performed by linking a 1-node heap to an existing heap, the complexity of an insert operation is O(1). Deletion is more complicated, entailing extraction of the highest priority event in the root, followed by a carefully chosen tree-pairing strategy combined with the linking operation. One pass is made to create the pairs, and a second pass creates the tree. For an n-event heap, the amortized cost of a delete operation has been shown²⁷ to be $O(\log n)$.

Calendar queue

The calendar queue¹¹ is a recently proposed simulation calendar implementation. Unlike the other self-adjusting structures described above, it is based on a multiple list structure. The event calendar consists of buckets called *days*, with a variable number of days comprising a *year*. Each day contains a number of events scheduled for that day, in sorted order (e.g. a sorted linked list).

The insert operation determines which bucket a scheduled event falls into based on current calendar parameters, and the scheduled time of occurrence of the event. Within a bucket, the event is inserted in sorted order. Since the calendar queue is intended to operate with a small, fixed number of events in each bucket, the complexity of an insert operation has experimentally been shown to be O(1). The delete operation only involves bucket determination, and hence is also O(1).

Brown¹¹ suggests that queue operation will be inefficient if queue size n is much smaller or larger than the number of buckets. To keep insertion and deletion times constant at all times, it is necessary for calendar parameters to be readjusted periodically. That is, the number of buckets is allowed to grow and shrink with calendar size. Unfortunately, this periodic readjustment is an O(n) operation. It can have a detrimental effect on overall performance if used too frequently.

REFERENCES

- 1. Y. F. Lam and V. K. Li, 'An improved algorithm for performance analysis of networks with unreliable components', *IEEE Trans. Communications*, **34**(5), 496–497 (1986).
- 2. S. Olariu and Z. Wen, 'Optimal parallel initialization algorithms for a class of priority queues', *IEEE Trans. Parallel and Distributed Systems*, **2**(4), 423–429 (1991).
- 3. J. B. Evans, Structures of Discrete Event Simulation, Ellis Horwood Ltd., Chichester, England, 1988.
- 4. A. Pritsker and C. Pegden, Introduction to Simulation and SLAM, 3rd edn, Wiley, New York, 1979.
- H. Kobayashi, Modeling and Simulation: An Introduction to System Performance Evaluation Methodology. Addison-Wesley Publishing Co., New York, 1981.
- 6. J. C. Comfort, 'The simulation of a microprocessor based event set processor', *14th Annual Simulation Symposium*, SCS & IEEE, Piscataway, N.J., 1981, pp. 17–34.
- 7. A. M. Law and W. D. Kelton, Simulation Modeling and Analysis, 2nd edn., McGraw-Hill, New York, 1990.
- 8. W. McCormack and R. G. Sargent, 'Analysis of future event set algorithms for discrete event simulation', *Communications of the ACM*, **24**(12), 801–812 (1981).
- 9. J. G. Vaucher and P. Duval, 'A comparison of simulation event list algorithms', *Communications of the ACM*, **18**(4), 223–230 (1975).
- 10. J. Henriksen, 'An improved event list algorithm', Winter Simulation Conference, IEEE and SCS, Piscataway, N.J., 1983.
- 11. R. Brown, 'Calendar queue, a fast O(1) priority queue implementation for the simulation event set problem', *Communications of the ACM*, **31**(10), 1220–1227 (1988).
- 12. D. W. Jones, 'An empirical comparison of priority-queue and event-set implementations', *Communications of the ACM*, **29**(4), 300–311 (1986).
- 13. R. Conway, B. Johnson and W. Maxwell, 'Some problems of digital systems simulation', *Management Science*, **6**(1), 92–110 (1959).
- 14. G. M. Birtwistle, O. Dahl, B. Myhrhaug and K. Nygaard, Simula Begin, Studentlitteratur, Lund, 1973.
- 15. J. G. Vaucher, 'On the distribution of event times for the notices in a simulation event list', *INFOR*., **15**(2), 171–182 (1977).
- 16. W. R. Franta and K. Maly, 'A comparison of heaps and the TL structure for the simulation event set', *Communications of the ACM*, **21**(10), 873–875 (1978).
- 17. G. H. Gonnet, 'Heaps applied to event driven mechanisms', *Communications of the ACM*, **19**(7), 417–418 (1976).
- 18. J. C. Comfort, 'A taxonomy and analysis of event set management algorithms for discrete event simulation', 12th Annual Simulation Symposium, SCS & IEEE, 1979.

- 19. G. Johnson, SIMSCRIPT II.5 User's Manual (Release 8), CACI Inc., 1974.
- W. McCormack and R. G. Sargent, 'Comparison of future event set algorithms for simulations of closed queueing systems', in N. Adam and A. Dogramaci (eds), *Current Issues in Computer Simulation*, Academic Press, NY, 1979.
- 21. W. McCormack, 'Analysis of future event set algorithms for discrete event simulation', *Ph.D. Thesis*, Syracuse University, 1979.
- 22. J. Vuillemin, 'A data structure for manipulating priority queues', *Communications of the ACM*, **21**(4), 309–314 (1978).
- G. Francon Jr., G. Viennot and J. Vuillemin, 'Description and analysis of an efficient priority queue representation', 19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, IEEE, Piscataway, N.J., 1978.
- 24. D. D. Sleator and R. E. Tarjan, 'Self-adjusting heaps', SIAM J. Computing, 15(1), 52-69 (1986).
- 25. D. D. Sleator and R. E. Tarjan, 'Self-adjusting binary trees', *Proc. ACM SIGACT Symposium on Theory of Computing*, ACM, New York, 1983.
- 26. M. L. Fredman, R. Sedgewick, D. D. Sleator and R. E. Tarjan, 'The pairing heaps: a new form of self-adjusting heap', *Algorithmica*, 1, (1), 111–129 (1986).
- T. J. Stasko and J. S. Vitter, 'Pairing heaps: experiments and analysis', Communications of the ACM, 30(3), 234–249 (1987).
- 28. R. Engelbrecht-Wiggans and W. Maxwell, 'Analysis of the time indexed list procedure for synchronization of discrete event simulations', *Management Science*, **24**(13), 1417–1427 (1978).
- F. P. Wyman, 'Improved event-scanning mechanisms for discrete event simulation', Communications of the ACM, 18(6), 350–353 (1975).
- 30. C. M. Reeves, 'Complexity analyses of event set algorithms', The Computer Journal, 27(1), 72–79 (1984).
- 31. J. Kingston, 'Analysis of algorithms for the simulation event list', *Ph.D. Thesis*, Department of Computer Science, University of Sydney, 1984.
- 32. J. H. Kingston, 'Analysis of Henriksen's algorithms for the simulation event list', SIAM J. Computing, 15(3), 887–902 (1986).
- 33. K. Chung, J. Sang and V. Rego, 'Sol.es: an object-oriented platform for event-scheduled simulations', 1993 Summer Computer Simulation Conference, SCS, Boston, MA., 1993.
- 34. V. Rego and V. Sunderam, 'Experiments in concurrent stochastic simulation: the EcliPSe paradigm', *J. Parallel and Distributed Computing*, **14**, 66–84 (1992).
- 35. V. Sunderam and V. Rego, 'EcliPSe: a system for high performance concurrent simulation', *Software—Practice and Experience*, **21**, 1189–1219 (1991).
- 36. K. Nygaard and O. Dahl, 'The development of the SIMULA languages', ACM SIGPLAN Notices, 13(8), 245–272 (1978).
- 37. H. Schwetman, CSIM User's Guide (Revision 16), MCC, Austin, Texas, 1992.
- 38. J. Henriksen and R. Crain, *GPSS/H User's Manual*, 2nd edn, Wolverine Software Corporation, Annandale, VA, 1982.
- 39. E. Cinlar, Introduction to Stochastic Processes, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1975.
- 40. A. Jonassen and D. E. Knuth, 'A trivial algorithm whose analysis isn't', *J. Comput. System Sci.*, **16**, 301–332 (1978).
- 41. W. Pugh, 'Skip lists: a probabilistic alternative to balanced trees', Communications of the ACM, 33(6), 668–676 (1990).
- 42. G.A. Davison, 'Calendar p's and queues', Communications of the ACM, 32(10), 1241-1243 (1989).
- W. Szpankowski and V. Rego, 'Some theorems on instability with applications to multiaccess protocols', Operations Research, 36, 956–966 (1988).
- J. D. Spragins, J. L. Hammond and K. Pawlikowski, *Telecommunications Protocols and Design*. Addison Wesley, New York, N.Y. 1991.
- 45. V. Rego and L. M. Ni, 'Analytic models of cyclic service systems and their application to token-passing local area networks', *IEEE Trans. Computers*, 37, 1224–1234 (1988).
- 46. J. H. Blackstone and G. L. Hogg, 'A two-list synchronization procedure for discrete event simulation', *Communications of the ACM*, **24**(12), 825–828 (1981).
- 47. J. W. J. Williams, 'Algorithm 232: heapsort', Communications of the ACM, 7, 347-348 (1964).
- 48. D. B. Johnson, 'Priority queues with update and finding minimum spanning tree', *Information Process Letter*, **4**, (3), 53–57 (1975).
- 49. C. A. Crane, 'Linear lists and priority queues as balanced binary tree', *Technical Report STAN-CS-72-259*, Stanford University, 1972.

- 50. R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, 1983.
- 51. T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1989.