A Comparative Study of Parallel and Sequential Priority Queue Algorithms

ROBERT RÖNNGREN and RASSUL AYANI
Royal Institute of Technology (KTH), Stockholm, Sweden

Priority queues are used in many applications including real-time systems, operating systems, and simulations. Their implementation may have a profound effect on the performance of such applications. In this article, we study the performance of well-known sequential priority queue implementations and the recently proposed parallel access priority queues. To accurately assess the performance of a priority queue, the performance measurement methodology must be appropriate. We use the Classic Hold, the Markov Model, and an Up/Down access pattern to measure performance and look at both the average access time and the worst-case time that are of vital interest to real-time applications. Our results suggest that the best choice for priority queue algorithms depends heavily on the application. For queue sizes smaller than 1,000 elements, the Splay Tree, the Skew Heap, and Henriksen's algorithm show good average access times. For large queue sizes of 5,000 elements or more, the Calendar Queue and the Lazy Queue offer good average access times but have very long worst-case access times. The Skew Heap and the Splay Tree exhibit the best worst-case access times. Among the parallel access priority queues tested, the Parallel Access Skew Heap provides the best performance on small shared memory multiprocessors.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**] Concurrent Programming; E.1 [**Data Structures**]: Lists, Trees; E.2 [**Data Storage Representation**]: Linked Representations; F.2.2 [**Analysis of Algorithms**] Nonnumerical Algorithms and Problems—*sequencing and scheduling*; I.6.8 [**Simulation and Modeling**]: Discrete Event

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Parallel access priority queue, pending event set implementations, priority queue

This work is part of a distributed simulation project financed by the Swedish National Board for Industrial and Technical Development (NUTEK). Part of this article is based on "Fast Implementations of the Pending-Event Set," by Rönngren, Riboe, and Ayani, in *Proceedings of The International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Part of the SCS Western Multiconference on Computer Simulation, San Diego, California, January 17–20, 1993 SCS.

Authors' addresses: Department of Teleinformatics, Simulation Laboratory, Royal Institute of Technology (KTH), Stockholm, Sweden; email: ⟨robertr,rassul@it.kth.se⟩ http://www.it.kth.se/labs/sim.

1. INTRODUCTION

Priority queues are used in a wide variety of applications including operating systems, real-time systems, and discrete event simulations. In a priority queue, each element is ordered by its associated priority. The basic operations are dequeue and enqueue. A dequeue operation removes the element with the highest priority, and an enqueue inserts a new element into the queue. The way the operations are performed may have a profound effect on the performance of such applications. Although priority queues are used in various contexts, there are some general quality measures of interest. The most important metric is the time required to perform the most common operations, in our case, dequeue and enqueue, and we refer to this time as *access time*. In most cases, the measure of interest is the amortized (or average) access time [Tarjan 1985]. However, for some applications (e.g., real-time systems), the maximum or worst-case access time is also of interest. Other characteristics that may influence the choice of a priority queue implementation are memory requirements, code size, and the possibility of providing additional operations such as retrieval of an arbitrary element.

Over the years, several performance studies on priority queues have appeared in the literature. A significant number of these studies have been performed in the context of discrete event simulation (DES).[1] Here, a priority queue is used to hold the pending event set (sometimes referred to as the event calendar [Chung et al. 1993]) which contains the generated but not yet evaluated events. The reason for studying priority queues in this context is twofold: the specific implementation is often crucial to the performance of the simulator, and the way operations are performed on the pending event set provides an excellent test case for studying priority queues. The former is accentuated in parallel discrete event simulation [Fujimoto 1990], where the impact of the implementation of the pending event set can have a superlinear effect on performance [Rönngren et al. 1993b]. Thus it is important to find methods that allow a realistic and accurate assessment of the access time.

Up till now, the most widely used method for performance studies of priority queues has been the Hold model introduced by Vaucher and Duval [1975] and refined by Jones [1986]. It models operations on a fixed-size queue where a series of hold operations (a dequeue followed by an enqueue) are performed. In the following, we refer to this model as the Classic Hold. This method, however, does not capture the dynamic nature of queue sizes that often appears in practice, as recognized by several researchers including Rönngren et al. [1993a] and Chung et al. [1993]. An Up/Down model is proposed in Rönngren et al. [1993a], where a sequence of enqueues is followed by an equally long sequence of dequeues. Chung et al. [1993] propose an elegant generalization of the Hold model, the Markov Hold,

[1] See Chung et al. [1993], Jones [1986], McCormack and Sargent [1981], and Vaucher and Duval [1975].

where operations on the queue are determined by a two-state Markov process with states insert (enqueue) and delete (dequeue). By changing the transition probabilities, the Markov Hold model can represent random sequences of enqueue and dequeue operations. Apart from queue size and access pattern, a priority queue may also be sensitive to the distribution function used to generate priority for the elements. This issue has been addressed in several studies.[2]

Recently, several new data structures for the implementation of priority queues have appeared in the literature.[3] We compare the performance of some of these data structures with data structures that have fared best in earlier studies. Two categories of priority queues are studied: general purpose priority queues and those tailored to discrete event simulation.

In this article, we present the results of an extensive experimental study of priority queues. The study encompasses several recently proposed data structures for the implementation of priority queues as well as data structures that have fared well in earlier studies. Our main contributions can be summarized as follows.

(1) We use three measurement methods: Classic Hold [Vaucher and Duval 1975; Jones 1986], Markov Hold [Chung et al. 1993], and Up/Down [Rönngren et al. 1993a] and comment on the advantages and disadvantages of each.

(2) For most of the experiments, we measure the access time of each individual enqueue/dequeue operation to isolate the effects of the underlying hardware. Most of the other studies have measured the access time by measuring the execution time of a loop consisting of a number of operations. Our method not only allows measuring the average access time, but also identifying the best and worst-case access times which are of great interest in real-time systems.

(3) We use some compound distribution functions which interleave sequences of time-stamp increments drawn from different distribution functions in addition to those used by other researchers. These distributions are of practical interest in applications such as traffic systems.

(4) We compare the performance of parallel access priority queue algorithms that have been recently proposed. To our knowledge, this has never been done before.

The remainder of this article is organized as follows. Section 2 provides an introduction to discrete event simulation and the use of priority queues for implementation of the pending event set; Section 3 describes performance measurement techniques; Section 4 gives a brief overview of the priority queues investigated in this article; results from the performance

[2] For example, Jones [1986], McCormack and Sargent [1981], Rönngren et al. [1993a], and Vaucher and Duval [1975].

[3] See Ayani [1990], Brown [1988], Jones [1989], Pugh [1990], Rao and Kumar [1988], and Rönngren et al. [1993a].

measurements are presented in Section 5; and conclusions and recommendations are found in Section 6.

2. PRIORITY QUEUES AND DISCRETE EVENT SIMULATION

In the unifying framework that we use to describe both sequential and parallel discrete event simulation, a system is modeled as a number of concurrent logical processes (LPs) interacting by scheduled event messages. The pending event set (PES) is the set of all generated but not yet evaluated events and, in general, is represented by a priority queue. The implementation of the PES is often crucial to simulation performance. An empirical study by Comfort [1984] indicated that up to 40% of the simulation execution time may be spent on the management of the PES alone. Therefore, as systems become more complex and a demand for fast simulators arises efficient implementation of the PES becomes increasingly important.

One particular problem lies in the size of the PES. In general, larger PESs result in slower execution time. Although most simulation models generate event sets of at most a few hundred events, some models can generate event sets containing thousands of events. In some cases, this problem can be minimized depending on the purpose of the simulation. If the purpose is to study statistical entities, it is sometimes possible to apply modeling techniques that simplify the model or to manipulate the statistical properties of the model to reduce the PES's size [Kesidis and Walrand 1993; Obal and Sanders 1994]. However, these methods are not generally applicable, and it may be hard to validate and scale the obtained results. Furthermore, as simulation is a widely used tool, users may create simulation models without being aware of the size of the event set that is generated. Other situations may occur where it is not possible to reduce the size of the PES. If the aim is to validate or verify the behavior of a system, then the model often has to be in great detail. This may, in turn, result in very large PESs and hence greater execution time. An example is gate-level simulation for validation of custom-design digital logic circuits (ASICs). Such systems are often built from several ASICs, where a single ASIC may consist of more than 100,000 gates. If only a fraction of the gates are simultaneously active, the PES can contain several thousand events. Other examples of simulations generating potentially very large event sets can be found in communication systems such as Personal Communication Systems [Carothers et al. 1994; Das et al. 1994] and telephone networks [Unger et al. 1994]. Thus a data structure suitable for the implementation of the PES in discrete event simulation systems should ideally be able to handle event set sizes ranging from a few to several thousand events efficiently.

2.1 Pending Event Set in Sequential DES

In a PES, the simulated times at which the events are scheduled to be executed (time-stamps) are used as priorities. A sequential discrete event simulator operates in a three-step cycle: remove the event with the smallest timestamp (i.e., highest priority) from the PES; execute this event; and insert any new events resulting from this execution into the PES. Thus the two most common operations on the PES data structure are: dequeue, the removal of the event with the highest priority (sometimes referred to as delete-min), and enqueue, the insertion of a new event. Empirical studies of real simulations [Comfort 1984] indicate that these two operations can account for as much as 98% of all operations on the PES, the rest being other operations such as deletion of arbitrary events and the like. The performance of the PES is influenced by a number of variables including the initial distribution of events, the priority (or time-stamp) increment distributions, access patterns (i.e., mixture of dequeue and enqueue operations), and the size of the event set [Jones 1986]. Thus the event set implementation must be efficient under a wide variety of operating conditions and possibly be adaptive to take advantage of these conditions [Jones et al. 1986].

2.2 Event Set in Parallel DES

The requirements of high performance simulation of complex systems and the observation that these systems are often inherently parallel have motivated the development of parallel (or distributed) discrete event simulation [Fujimoto 1990; Jeffersson 1985; Misra 1986]. In parallel DES (PDES), the inherent parallelism that exists in most simulation models is realized by allowing LPs to be executed in parallel using several processing units. From a conceptual view, the LPs are independent self-contained processes. Among other things, this implies that the global PES is often divided over the LPs so that each LP has its own input queue of events (or event list). In PDES priority queues are also often used as scheduling queues for LPs that are ready to execute (i.e., LPs that have events to execute). This queue may also be shared by several processing elements.

2.3 Stability and Determinism

An issue that is important in many cases but is often neglected when using priority queues is how elements with identical priorities are handled. In DES, the user often assumes that if two events with identical time-stamps are inserted into the PES in a specific order, these events will be dequeued in the same order (i.e., FIFO order). A priority queue that preserves FIFO order on items with equal priority is referred to as *stable* [Gordon 1981]. A stable PES is often desirable as it may facilitate debugging of a simulation model/program since all events will be evaluated in the order in which they were generated. Nevertheless, many popular priority queue algorithms are not stable, such as the Implicit Binary Heap. Nonstable priority queues can be made stable by the introduction of an auxiliary priority field, a sequence number [McCormack and Sargent 1981], to break ties. Although stability is often considered to be critically important we should, for the sake of completeness, note that there exist different opinions on this issue. One counterargument is that (implicit) assumptions on the ordering of events with identical time-stamps as a basis for the correctness of a simulation model in some cases can make the model harder to understand and difficult to maintain.

A related concept to stability is *determinism*, which means that two runs of a simulation model with identical parameter settings produce the same results. Determinism is an important property of a simulation system as it facilitates debugging of the code as well as validation of the model. A simulator is deterministic if all LPs in a model execute their events in the same order in any two runs with identical parameter settings [Jeffersson 1985]. Determinism is inherent in a sequential simulation but not in a parallel one.

A parallel simulation is considered to be correct if all LPs process all events in correct causal order [Jeffersson 1985]. However, this does not guarantee that the simulation is deterministic. The real-time order in which two independent events are generated by two different processing elements may vary for different runs of a simulation. Hence a PDES cannot rely on a real-time ordering of events to achieve deterministic results. Enforcing determinism in a parallel simulation requires the ability of imposing a total ordering of the events. Methods by which events in distributed systems can be ordered have been proposed by Lamport [1978] and for PDES in particular by Mehl and Hammes [1993].

3. PERFORMANCE MEASUREMENT TECHNIQUES

As indicated in Sections 1 and 2, the PES must be able to operate under a wide variety of operating conditions. When designing experiments to study priority queues, it is important to carefully choose access patterns, priority increment distributions, and measurement techniques. The choices should reflect the operating conditions under which the priority queue will be used as well as enabling accurate measurement of the performance metrics of interest. In this section we describe the methods that have been used and evaluated in this study. It also contains a description of the computer and programming systems used in the experiments.

3.1 Access Patterns

When selecting access patterns for this study we chose synthetic experiments [Chung et al. 1993; Jones 1986; Vaucher and Duval 1975] over real simulations [Chung et al. 1993; McCormack and Sargent 1981]. Synthetic experiments provide better control over the variables affecting performance and, thus, they better expose the factors that influence performance. Furthermore, synthetic experiments facilitate direct comparison to earlier priority queue studies [Chung et al. 1993; Jones 1986; Vaucher and Duval 1975].

Three classes of accesses can be identified: steady-state, transient behavior, and random access patterns. The steady-state is the most commonly used method to study priority queues and is primarily modeled as Classic

Hold.[4] A hold operation is defined as a dequeue followed by an enqueue operation. Up/Down experiments proposed by Rönngren et al. [1993a] model the PES's transient behavior where the queue grows to a certain size by a sequence of enqueues and then shrinks by a sequence of dequeues. In our experiments, the measurements start and end with an empty queue for this access pattern. In a random access pattern, each access may be a dequeue or an enqueue operation.

In simulation experiments it is often necessary to let the simulation run for some time before statistics are collected. This is to avoid the transient startup period of the simulation. Similar transient periods occur in the synthetic experiments before the distribution of events in the event set has stabilized. This phenomenon has been analytically studied in the context of the hold model by Vaucher [1977] and Steinman [1994], respectively. These studies show that the distribution of the events in the queue under the hold model eventually reaches a steady state that is entirely determined by the priority increment distribution. Given these distributions, it is possible to calculate a measure on the average fraction of the events that an enqueue operation would scan in a linked list implementation of the PES after an initial phase of an infinite number of hold operations [Kingston 1985; McCormack and Sargent 1981]. In this article, we use the term *bias*, as suggested by Jones [1986], for this measure. A bias value of 1 corresponds to a pure FIFO behavior and similarly, a value of 0 corresponds to a LIFO queue. McCormack and Sargent [1981] observed that in practice different LPs often use different priority increment distributions. They also showed that the mixing of priority increment distributions tends to result in low bias. To model this phenomenon, they proposed an interaction hold model where the priority increment is drawn randomly from a set of distribution functions. Kingston [1985], however, showed that this particular model is, in general, equivalent to the Classic Hold model using a single priority increment distribution.

The Classic Hold models the behavior of a discrete event simulation system performing a sequence of hold operations. In the Classic Hold experiments, the queue is initialized to a certain size by a stochastic series of enqueue and dequeue operations with a slightly higher probability for enqueue operations than dequeue operations [Jones 1986]. When the queue has reached the desired size, the measurement phase which consists of a number of hold operations is performed. The manner in which the buildup phase of the queue is performed ensures that the shape of the tree and heap-structured queues will be correct [Jones 1986]. After the buildup phase, the distribution of the events in the queue undergoes a transient period before bias has reached the theoretical value. The length of this transient period, which is often short [McCormack and Sargent 1981], depends on the number of events in the queue and the priority increment distribution. Measuring the average bias from the first hold operation after

[4] See Jones [1986], Kingston [1985], McCormack and Sargent [1981], and Vaucher and Duval [1975].

$$\begin{array}{c|cc|}
 & \text{Dequeue} & \text{Enqueue} \\
\text{Dequeue} & \alpha & 1-\alpha \\
\text{Enqueue} & 1-\beta & \beta
\end{array}$$

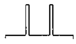
Fig. 1. State transition probability matrix for Markov Hold.

the buildup phase, we have observed that for priority increment distributions with bias greater than or equal to 0.5, the measured bias will, in general, converge to the theoretical value after a number of hold operations equalling 5 times the queue size. For priority increment distributions with bias less than 0.5, the required number of hold operations is, however, often substantially longer, up to 30 times the queue size. Thus, if access time measurements begin directly after the buildup phase, we expect that correct average values for the access time will be obtained after a number of hold operations equal to 5 or 30 times the queue size. This method to determine the measurement phase has the advantages (i) that the problem of determining the length of the transient period is avoided, and (ii) that the transient period will influence the performance measurements for different queue sizes to the same extent. The latter would not necessarily be the case if the same number of operations had been used for the observation and/or skipped transient periods for all queue sizes.

The Classic Hold and the Up/Down models represent two extreme cases and serve to show the performance bounds of PES implementations. Experiments were performed for various queue sizes ranging from 25 to 50,000 elements. In the Classic Hold experiments, the number of hold operations performed was 5 times the queue size for all experiments performed on the Sequent Symmetry (due to prohibitively long execution times for the larger queue sizes) and 30 times the queue size on the SUN Sparc10 and PentiumPro PC (see Section 3.3).

Chung et al. [1993] proposed an elegant generalization of the Classic Hold, called the Markov Hold model. In Markov Hold, the operations on the priority queue are generated by a two-state Markov process that may be in either of the states insert (enqueue) or delete (dequeue). By changing the state transition probabilities (α and β in Figure 1) this model can be used to represent the Classic Hold, transient behavior, and a generalized random sequence of enqueue and dequeue operations. Chung et al. argued that in real-life simulations, the effects of changing queue sizes will dominate over the effects of using different priority increment distributions. However, according to our experience, this only holds if the considered priority queue implementation is relatively insensitive to the particular priority increment distribution function used (see the discussion in Section 3.2). With this reasoning, Chung et al. [1993] used only a mixture of two priority distributions in their reported experiments. With an enqueue immediately followed by a dequeue, the priority increment is drawn from an Erlang distribution with mean 2 and standard deviation 1. In all other cases, the priority increment is drawn from a two-component hyperexponential distri-

Table I. Priority Increment Distributions

Distribution	Expression to compute random number	Bias
1 Exponential(1)	**if** (x = rand()) == 0 **then** infinity **else** -ln(rand())	0.50
2 Uniform(0,2)	2*rand()	0.66
3 Uniform(0.9, 1.1)	0.9 + 0.2*rand()	0.96
4 Bimodal	9.95238*rand() + **if** rand()<0.1 **then** 9.5238 **else** 0	0.34
5 Triangular (0, 1.5)	1.5*sqrt(rand())	0.80
6 Negative triangular(0,1000)	1000*(1-sqrt(1-rand()))	0.60
7 Camel (0.001,0.999,0,1000,2)	 See [Rönngren et al. 1993a; Riboe 1991]	0.74
8 ExponentialMix	**if**(rand() < 0.9) **then** Exponential(1) **else** Exponential(100)	0.10

rand() returns a uniform random number in the interval [0,1]

bution with mean 2 and standard deviation 4. As a consequence of this, only the Erlang distribution is used in a sequence of hold operations as in Classic Hold, and the hyperexponential distribution for a series of consecutive enqueues.

Although the Markov model can be used to generate access patterns equivalent to the three classes mentioned (Classic Hold, transient behavior, and generalized random sequence of enqueue and dequeue operations), we also studied these models to: (i) compare our results with the published works which are mainly based on the Classic Hold, and (ii) evaluate the Markov model for a wide variety of parameters.

3.2 Priority Increment Distributions

The priorities (time-stamps) of new elements are calculated by adding an increment generated by a priority increment distribution function to the value of the most recently dequeued element. Different distributions can result in access patterns ranging from near FIFO to near LIFO behavior. Moreover, some queue implementations are sensitive to the shape of the distributions.

The distributions used in this study are found in Table I, where rand( ) returns a random number in the interval [0, 1], as described in Park and Miller [1988]. Several of these distributions have been used in other studies.[5] The bias values have been measured in separate experiments and correspond well with analytically calculated and measured values which have been presented in earlier studies, with the exception of the bimodal

[5] See Brown [1988], Henriksen [1977], Jones [1986], Kingston [1985, 1986], and McCormack and Sargent [1981].

distribution, reported in Jones [1986]. Jones reports that this distribution should have a bias value of 0.13. This distribution is, however, very similar to several of the distributions used by McCormack and Sargent [1981] (i.e., distributions 1, 13, and 14) for which the reported bias values fell in the range 0.17 to 0.42. We therefore believe the value 0.34 to be correct for this distribution. The last three distributions of Table I, the negative triangular, the camel distribution [Rönngren et al. 1993a; Riboe 1991], and the ExponentialMix are specifically intended to test some of the special purpose PES implementations. The camel distribution may be used to model aspects of bursty traffic in computer and telecommunication networks. The parameter setting used for the camel distribution results in a double hump distribution. A 0.001 fraction of the probability mass is evenly distributed over the interval [0, 1,000] and the remaining 0.999 fraction is concentrated in two equally large humps located at one-third and two-thirds from the beginning of the interval. The two humps have each a width of 0.0005 of the total interval length. The ExponentialMix distribution is a slight variation of the distribution labeled 3 in McCormack and Sargent [1981].

The use of a single distribution does not always reveal all weaknesses of a particular priority queue implementation. Some of the special-purpose PES implementations that depend on resize heuristics and operations, such as the Calendar queue and the Lazy Queue, may be sensitive to drastic changes in priority increment distribution [Rönngren et al. 1993a]. For this reason, we have tested the queues with interleaving sequences of priority increments drawn from two different distributions. This is defined as Change(A, B, x), where A and B are priority increment distributions and x is an integer value. The initial x priority increments are drawn from distribution A, the following x increments are generated by distribution B, the next x increments from distribution A, and so on [Rönngren et al. 1993a].

3.3 Coding Conventions and Target Machine

The experiments measuring the performance of the priority queues were primarily conducted on a Sequent Symmetry[6] S81 [Sequent 1989] shared memory bus-based multiprocessor equipped with 10 Intel 386i processors operating at 16 MHz (delivering 1.6–2.4 Mips per processor). The machine is equipped with 40 Mb shared RAM, 20 MHz external floating point units, and 32 Kb caches for data and instructions using a copyback cache consistency protocol. The bus has a bandwidth of 53.3 Mb/s. Although the Sequent Symmetry S81 does not have impressive performance by today's standards, it provided us a single platform for both parallel and sequential experiments and allowed a more accurate time measurement technique than could be accomplished on more modern computers (see Section 3.4).

Our measure for the performance of the priority queues is execution time. This measure is accurate for the particular hardware configuration on

[6] Symmetry is a registered trademark of the Sequent Computer Systems, Inc.

which it has been obtained. However, the characteristics of different computer architectures may affect the execution speed of different priority queues differently, thereby changing their relative performance. To give an indication of whether conclusions based on the Sequent Symmetry experiments change when more recent computer architectures are used, we conducted additional experiments on a SUN Sparc10 workstation and a high performance PC. Both of these computers used the Solaris 2.5 operating system and were equipped with two processors and 64 Mb memory. The Sparc10 was equipped with 40 MHz processors and the PC had PentiumPro 200 MHz processors.

All code was written in the C programming language and optimized by eliminating all recursive procedure calls, and the like. Four operations on the queues were implemented: enqueue, dequeue, create-queue, and destroy-queue. For some of the queues, it is possible to implement a specific hold operation that is more efficient (see McCormack and Sargent [1981]). However, this was not done since the practical importance of such an operation is limited, and it would make our comparison more complicated. No assumptions on the queue size were made when the queues were created.

The performance measurements were performed with the needed memory pre-allocated (when necessary managed in free-lists). Thus the effects of the underlying memory management system on the performance were minimized. In the parallel experiments, the locks and the macrolock operations provided by the Sequent microtasking library were used. The average cost of accessing a lock was 4–5 μs.

3.4 Time Measurements

Two different measurement techniques were used. In the sequential experiments on the Sequent Symmetry S81 each operation on the queue was timed separately [Rönngren et al. 1993a] with the use of the microsecond resolution clock available on the Sequent Symmetry S81. This technique has the important property that time measurements are restricted to the queue accesses, which improves the accuracy of the performance measurements. This technique not only allows measurements of the amortized access times [Tarjan 1985], but also the worst-case access time.

For the other experiments, this technique was not feasible. On the SUN Sparc10 and the PentiumPro equipped PC, this was due to insufficient resolution of the available time measurement mechanisms. Measuring individual operations in the parallel experiments would disturb the potential parallelism. Instead, a more traditional two-loop technique was employed in these cases. One of the loops included the queue accesses and the other did not. The execution time corresponding to the first loop, T_1, consisted of the loop overhead and the queue access time, and the second loop time, T_2, represented only the loop overhead. Thus the total queue access time was $T_1 - T_2$, and the average access time was $T_{average} = (T_1 - T_2)$/Total number of operations.

4. PRIORITY QUEUE DATA STRUCTURES

Both sequential access and parallel access priority queue implementations are discussed in this work. In the following sections we discuss: (i) the sequential priority queue data structures evaluated in this study and their expected performance, (ii) particular issues related to parallel access priority queue algorithms, and (iii) the parallel access priority queue algorithms that we have selected for this study.

4.1 Sequential Priority Queue Data Structures

Our study includes: (i) the following well-known data structures: implicit binary heaps [Bentley 1985], linked lists, Skew Heaps [Sleator and Tarjan 1986], Splay Trees [Sleator and Tarjan 1985], Calendar Queue [Brown 1988], skip lists [Pugh 1990], and Henriksen's [1977] algorithm, previously studied by other researchers [Chung et al. 1993; Jones 1986; McCormack and Sargent 1981] and (ii) some of the more recently proposed queue implementations, that is, the Lazy Queue [Rönngren et al. 1993a] and the SPEEDESQ [Steinman 1992]. The Calendar Queue, Henriksen's algorithm, the SPEEDESQ, and the Lazy Queue are specifically tailored to take advantage of the special characteristics of the PES in discrete event simulation.

Of these data structures, the Median Pointer Linked list, the Calendar Queue, the skip lists, the Splay Tree, and Henriksen's algorithm are stable. The Lazy Queue and the SPEEDESQ can be made stable if the sorting algorithms used in the implementation of these queues are stable. In this study we have, however, not used stable sorting algorithms for these queues. Moreover, our implementations of the implicit binary heap and the Skew Heap are not stable; that is, they do not implement auxiliary sequence numbers [McCormack and Sargent 1981]. The characteristics of these data structures are discussed in the following.

4.1.1 *Implicit Binary Heap.* The sequential implicit binary heap [Bentley 1985] (referred to as implicit 2-heap in Chung et al. [1993]) is well known, and the data structure as such should need no further introduction. However, there is one observation worth making on the computational complexity of the implicit binary heap. Both the enqueue and dequeue operation have a worst-case behavior of $O(\log(n))$, where n is the number of elements in the queue. The amortized complexity of enqueue operations is, however, often near $O(1)$ in practical cases. In an enqueue operation, the new element is placed at the base of the heap (i.e., as the rightmost leaf at the lowest level). To restore the heap property (i.e., that each parent has a higher priority than any of its children), the new element is compared to its parent and if necessary they are swapped. This process has to be repeated upwards in the heap until either the root is reached or at some level no swap is needed. The question is then, how many levels have to be traversed on average? This problem can be answered by calculating at what level (enumerating the levels 1, 2, 3, . . . from the bottom-most level) an arbitrary element is placed. Consider a heap where the last level is fully filled, then half the elements are at level 1, one quarter of the elements are at level 2, and so on. Thus the average level can be expressed as the sum 1*1/2 + 2*1/4 + 3*1/8 + . . . , which is asymptotically bounded by 2. Consequently, if the bias is not extremely low we would expect the amortized enqueue time to be near O(1).

4.1.2 *Median Pointer Linked List.* The Median Pointer Linked list was implemented as a doubly linked circular list that allows insertions to take place from both the front and the back. A pointer to the median element (with respect to the number of elements) in the list is used to identify whether the insertions should be made from the front (time-stamp of the new element is less than or equal to that of the median element) or the back end of the list [McCormack and Sargent 1981]. A dequeue operation on a Median Pointer Linked list is performed in constant time, whereas the enqueue operation is O(n).

4.1.3 *Skew Heap.* The Skew Heap [Sleator and Tarjan 1986] is a heap-ordered binary tree where any descendant of a node has lower priority than the node itself. The central operation in the Skew Heap is referred to as a meld operation. A meld operation merges two Skew Heaps into one, preserving the heap property. Thus a dequeue operation is performed by removing the topmost (root) node and melding the two resulting subheaps into a single heap. In the top-down Skew Heap used in our study, an enqueue operation is performed as a meld of a one-node Skew Heap and the existing Skew Heap. Our implementation was based on the nonrecursive code found in Jones [1989]. The amortized time to perform a dequeue or an enqueue operation is O(log(n)) [Sleator and Tarjan 1986] although individual operations may be O(n) if the heuristics for the balancing of the heap fails.

4.1.4 *Splay Tree.* Splay Tree [Sleator and Tarjan 1985] is a heuristically balanced binary search tree. It uses a balancing technique called splaying. Splaying is essentially a sequence of tree rotations that helps balance the tree and move nodes on frequently used branches upwards (closer to the root). Our implementation was based on the Pascal code used by Jones [1986]. In this implementation, a dequeue operation can be performed in constant time and the amortized access time for the enqueue operation is O(log(n)) [Sleator and Tarjan 1985]. The worst-case access time of an individual enqueue operation is O(n) if the balancing heuristic fails.

4.1.5 *Calendar Queue.* The Calendar Queue suggested by Brown [1988] is a multilist-based data structure. It uses an elegant technique to manage the overflow problem encountered in multilist-based implementations. In the Calendar Queue, there exists no dedicated overflow structure. All elements, including those that would fall into an overflow structure in an ordinary multilist, are inserted into the sublists. This is accomplished in the following way: all sublists span equally long priority (time) intervals where the total length of these subintervals is called a year. When a new element is inserted in the queue, the sublist into which the new element will fall is calculated. To ensure good performance, it is required that the sublists, which are implemented as linked lists, are kept short (an average length of two elements). This is accomplished by a resize operation. The resize operation is performed when the queue size has changed by a factor of two. The new length of the subintervals is calculated by using an approximation of the distribution based on the first few elements [Brown 1988]. Our implementation is based on the calendar queue code supplied by Brown [1988]. The Calendar Queue has O(1) access time under many operating conditions although the resize operations are O(n). The worst-case amortized access time for the Calendar Queue is O(n) [Rönngren et al. 1993a].

4.1.6 *Lazy Queue.* The Lazy Queue [Rönngren et al. 1993a] is a multilist-oriented data structure. The fundamental idea is to divide the future events into several parts and keep only a small portion of the elements completely sorted. The elements are divided into: (i) a near future that is kept sorted, (ii) a far future that is partially sorted, and (iii) a very far future that is used as an overflow bucket. As time advances, part of the far future is sorted by standard sorting techniques and transferred into the near future. This lazy sorting behavior gives the queue its name. The far future part of the queue is implemented as an array of unsorted sublists where each sublist corresponds to an equally long priority interval. The near future consists of a sorted array of elements (transferred from the far future) and a skew heap is used to insert the new elements that belong to the near future. Skew heaps are used to implement the very far future of the lazy queue.

To ensure good utilization of the data structures, a set of resize operations was introduced. Some modifications in the implementations of the resize operations have been performed that improve the performance for smaller queue sizes as compared to the results presented in [Rönngren et al. 1993a]. In particular, there is a resize operation that recalculates both the length of a subinterval and the number of subintervals. This operation is used whenever a resize operation is initiated. However, the criteria for initiating a resize operation remain unchanged. The resize operations are expensive, worst case of O($n$log(n)), but they are amortized over the relatively inexpensive ordinary operations. This results in a near O(1) access time for many operating conditions. The worst-case amortized access time can be restricted to O(n) [Rönngren et al. 1993a]. A minimum queue size of 256 elements is requested to perform a resize operation. The Lazy queue is stable only if the sorting algorithm and the implementations of the near and very far futures are stable.

4.1.7 *Henriksen's Data-Structure.* Henriksen's [1977] implementation of the PES employed in GPSS/H [Gordon 1981] uses a linked list and an array of pointers into the list. The array of pointers is used to perform binary searches in the list to find the place where a new element should be put in an enqueue operation. A heuristic algorithm is employed to update the auxiliary pointers. The implementation was based on the code given in

Kingston [1986] where the array of pointers is used as a circular list of pointers. The size of the array of pointers is doubled when necessary as the queue size grows. However, the array is not decreased in size if the queue size shrinks. Our implementation does not allow insertions of new events with time-stamps less than that of the most recent event dequeued from the queue. The amortized access time of Henriksen's algorithm is often O(log(n)), and limited by O($n^{1/2}$) in the worst case [Kingston 1986]. Dequeue operations are performed in constant time, whereas enqueue operations can be O(n).

4.1.8 *Skip List.* Skip List was proposed as an alternative to balanced search trees by Pugh [1990]. The Skip List is built on an ordinary doubly linked linear list where the elements of the list may carry additional forward pointers to successors in the list. The "level" of an element decides how many forward pointers the element has. An element of level k has k forward pointers pointing to the next elements of level 1, 2, . . . , k, respectively. The level of an element is chosen randomly when the element is inserted in the list and the maximum level of a node is limited to some fixed value. The basic idea of the Skip List is that the additional pointers form a randomly balanced search tree on top of the list without the additional costs required to balance the tree. Our implementation of the Skip List was based on code retrieved via ftp according to the instructions in Pugh [1990]. The Skip List has a O(log(n)) behavior provided the probabilistic search tree remains nearly balanced. If this is not the case, enqueue operations may take O(n) time, whereas dequeue operations can always be performed in constant time.

4.1.9 *SPEEDESQ.* The SPEEDESQ [Steinman 1992] consists of two single linear linked lists. One list, referred to as the "dequeue-list," is kept sorted and the other list, the "enqueue-list" is unsorted. New elements are added to the enqueue-list, which can be done in constant time since the list is kept unordered. The queue also maintains a variable recording of the highest priority (smallest time-stamp) of any element present in the enqueue-list. A dequeue operation removes the element with the highest priority from the dequeue-list. The enqueue-list is sorted and merged with the dequeue-list in a dequeue operation if the dequeue-list is exhausted or whenever the highest priority element is present in the enqueue-list. The merge operation is potentially an O(n) operation which, if frequent, results in a worst-case performance of O(n). The SPEEDESQ resembles the two-list structure proposed by Blackstone et al. [1981]. However, the main difference lies in the way the elements are transferred from the enqueue-list to the dequeue-list. Whereas the SPEEDESQ transfers the complete enqueue-list, the two-list approach scans the enqueue-list and transfers only a subset of the elements, that is, those with the highest priority. The two-list selection of which elements to transfer is based on some heuristics. If these heuristics fail, there is the potential for frequent scans of a very long enqueue-list in which case the performance would deteriorate. In this perspective, the SPEEDESQ approach for handling the enqueue-list is an

Table II. Expected Performance of Sequential Priority Queues

Queue	Enqueue amortized (expected, worst case)	Enqueue max	Dequeue amortized (expected, worst case)	Dequeue max
Calendar Queue	O(1), O(n)	O(n)	O(1), O(n)	O(n)
Splay Tree	O(log(n))	O(n)	O(1)	O(1)
Henriksen's	O(log(n)), O($n^{1/2}$)	O(n)	O(1)	O(1)
Skew Heap	O(log(n))	O(n)	O(log(n))	O(n)
Lazy Queue	O(1), O(n)	O($n$log(n))	O(1), O(n)	O($n$log(n))
Implicit Binary Heap	O(1), O(log(n))	O(log(n))	O(log(n))	O(log(n))
SPEEDESQ	O(1)	O(1)	O(1), O(n)	O($n$log(n))
Skip Lists	O(log(n))	O(n)	O(1)	O(1)
Median Pointer Linked list	O(n)	O(n)	O(1)	O(1)

improvement that eliminates this heuristic. SPEEDESQ has constant enqueue time and a constant time for many of the dequeue operations. However, dequeue operations that involve sorting of the enqueue-list have an O($n$log(n)) time complexity.

4.1.10 *Expected Performance of Sequential Priority Queues.* Table II summarizes the theoretically expected performance of the sequential priority queues studied in this article. For some of the data structures, there are two figures in the columns for amortized enqueue and dequeue times. These figures correspond to the expected amortized access times for a majority of operating conditions encountered in simulations and to the worst-case amortized behavior, respectively. When the worst and expected cases are equal, we give only one figure in the table. In Section 5, we compare these theoretically expected values with our experimental results.

4.2 Parallel Access Priority Queues

When designing parallel algorithms, there are a number of design choices that have to be addressed in addition to what is encountered in sequential programming. In particular, the parallelism has to be exploited efficiently while ensuring that the algorithm gives correct results and is free from deadlocks and livelocks. This section describes the evolution of parallel access priority queues in the context of DES and the design choices that have been made.

It is potentially possible to perform enqueue operations in parallel if they operate on separate parts of the priority queue. Apparently, this was first recognized within the database community where examples of concurrent software operations on data structures similar to priority queues were proposed [Bayer and Schkolnik 1977; Ellis 1980]. The first approach (to the authors' knowledge) to parallelizing the operations on the PES was based on special-purpose hardware. Comfort [1984] discussed a special-purpose multiprocessor system with several coprocessors managing a PES. The result of this study indicated that substantial reductions in execution time can be achieved by performing parallel operations on the PES. However, the potential level of parallelism in these experiments appeared to be limited to three. A software solution to parallel access priority queues that allowed simultaneous update of the queue was proposed by Biswas and Browne [1987]. Apart from managing the PES, parallel access priority queues could be used for scheduling queues in parallel processing environments [Ahmed et al. 1994].

When designing a parallel access priority queue, one main question is whether the process accessing the queue (i.e., performing a dequeue or enqueue operation) should be responsible for reordering the queue or if special processes should perform this task. The latter approach is advocated in both Comfort [1984] and Biswas and Browne [1987]. The use of maintenance processes may help achieve better locality and better cache performance as the location of the pending event set could be confined to some processors.

More recent parallel access priority queues [Rao and Kumar 1988; Jones 1989; Ayani 1990] do not, however, use separate maintenance processes. Instead, the processes performing the dequeue/enqueue operations also perform the restructuring operations. However, this is logically equivalent to using maintenance processes. It can be viewed as if the process, after claiming the element with the highest priority (in a dequeue) or bringing a new element to the queue, transforms into a maintenance process to perform the restructuring of the queue. The advantage of this approach is the absence of an explicit context switch between the accessing process and the maintenance process. With this view, it is also evident that the queue accesses are serializable if a dequeue operation returns the element with the highest priority when the operation started.

4.3 Parallel Access Priority Queue Algorithms

In this study we considered parallel access versions of the binary heap [Rao and Kumar 1988], Skew heap [Jones 1989], Lazy Queue, and linked list implementation of priority queues. The question of stability in the context of parallel access priority queues is more complex than in the sequential case. The first observation is that the issue of stability (in the context of DES) is based on the notion that there exists a particular real-time order in which events are generated and that this order is the same for any two executions of the same simulation model with identical parameter settings. As pointed out in Section 2.3, this may not be the case in a parallel environment where priorities or time-stamps of events are often made unique to accomplish determinism, in which case the question of stability is no longer relevant. The second observation is that in all implementations described in this section, any access to the priority queue has to go through a single entry point protected by a lock. Thus a prerequisite for a stable queue is that the queueing strategy for several processes contending for the lock is order preserving. This may, however, be system-dependent. Given order-preserving locks, the linked list implementation is stable.

4.3.1 *Parallel Access Binary Heap.* In sequential implementations of the implicit binary heap, the heap is traversed level by level. In each step, at most one parent and its child node are affected (their contents may be swapped). In the conventional heap, the dequeue operation traverses the heap top-down, and the enqueue operation traverses the heap bottom-up. This method could potentially lead to deadlock in a parallel implementation. This problem was resolved by Rao and Kumar [1988], who devised an enqueue operation that traverses the heap in a top-down manner. Each node of the heap is also equipped with a lock and a flag to show whether the content of the node is present or pending.

4.3.2 *Parallel Access Skew Heap.* In the Skew Heap, the operations are performed in a top-down manner traversing the heap level by level. The operations involve at most a parent and two child nodes at a time. Thus the basis for a parallel implementation exists in which several processors could work in parallel while updating the heap. This was recognized by Jones [1989] who suggested a parallel access implementation based on a parallel meld algorithm where each node is equipped with a lock (a semaphore in the original algorithm). The original algorithm suggested by Jones was recursive. In the implementation used in our experiments, an iterative implementation was devised which further improved the performance.

4.3.3 *Parallel Access Lazy Queue.* In the Lazy Queue [Rönngren et al. 1993a], an operation first accesses the common descriptor, then it accesses one of the near future, the sublists of the far future, or the very far future. A dequeue operation generally accesses the near future whereas enqueue operations tend to access the far future or possibly the very far future. The time needed to access the descriptor is slightly shorter than the time spent in any of the "future" parts of the queue. This implies that the potential level of parallelism, that is, the maximal number of simultaneous accesses, is slightly higher than two. In the current parallel implementation, the descriptor and the near future have been equipped with one lock each. The far future and the very far future have been equipped with a common lock. This allows one dequeue and one enqueue operation to be overlapped with one more operation. The resize operations of the Lazy Queue have not yet been parallelized for the sake of simplicity, although they offer potential for further parallelism.

4.3.4 *Parallel Access Linked List.* In a parallel implementation of the linked list, performing enqueue operations from both the front and the back end of the queue may lead to deadlock. Hence, a single linked list where all operations traverse the list from the front end has been chosen as the basis for the parallel implementation. The implementation is straightforward; each node is equipped with one lock. In a dequeue operation, only the head of the list needs to be locked. An enqueue operation traverses the list locking at most two successive nodes at a time.

4.3.5 *Expected Performance of Parallel Access Priority Queues.* The parallel access priority queues in this study are, to some extent, serialized as all operations have to go through a critical section (e.g., the topmost node in a heap, the descriptor in a heap, and the head of the list in a linear list implementation). Thus this critical section restricts the performance of the parallel implementations.

The heap-based implementations have operations with a time complexity of O(log(n)), but they allow for a potential degree of parallelism of O(log(n)) to be exploited. The Lazy Queue has a near O(1) behavior in its sequential version. The linked list has an O(1) dequeue operation and the enqueue operation is O(n). On the other hand, it allows for an O(n) degree of parallelism. Thus all the parallel access implementations have a potential for a near O(1) behavior. Questions that have to be answered are: to what extent can the parallelism offered by the different implementations be exploited? Will the queue size affect the performance? Experimental results answering these questions as well as the results from the sequential experiments are found in the next section.

5. PERFORMANCE MEASUREMENTS

In this section we present the results of the performance measurements. The experiments revealed that all queue implementations, except the Median Pointer Linked list, are only marginally dependent on the seed used in the priority increment distributions. Several of the implementations were also less sensitive to the priority increment distribution, making, in many cases, the individual curves from different distributions hard to distinguish. The results presented are the average values from experiments with 10 different initial seeds. The relative error in these experiments is less than 4%. Note that for many of the figures, a logarithmic scale has been used for the queue-size axis leading to linear plots for logarithmic complexity.

5.1 Classic Hold Versus Up/Down Model

Experiments with the Classic Hold and Up/Down models were performed on the Sequent Symmetry, the SUN Sparc10, and the PentiumPro based PC. The reader should bear in mind that the number of operations performed for the Classic Hold model on the Sequent Symmetry equals 5 times the queue size whereas the corresponding figures for the other architectures are 30 times the queue size. This implies that the performance results for the bimodal priority increment distribution on the Sequent Symmetry essentially reflects a transient period with respect to bias (see discussion in Section 3.1).

5.1.1 *Performance on the Sequent Symmetry.* Figures 2 and 3 depict the performance of the Implicit Binary Heap [Bentley 1985]. As expected, the Implicit Binary Heap exhibits O(log(n)) performance and depends only marginally on the priority increment distribution.

The access time of the Median Pointer Linked list, Figures 4 and 5, grows linearly with the number of elements in the queue. Its performance quickly deteriorates as the queue size exceeds 100 elements. The Classic Hold

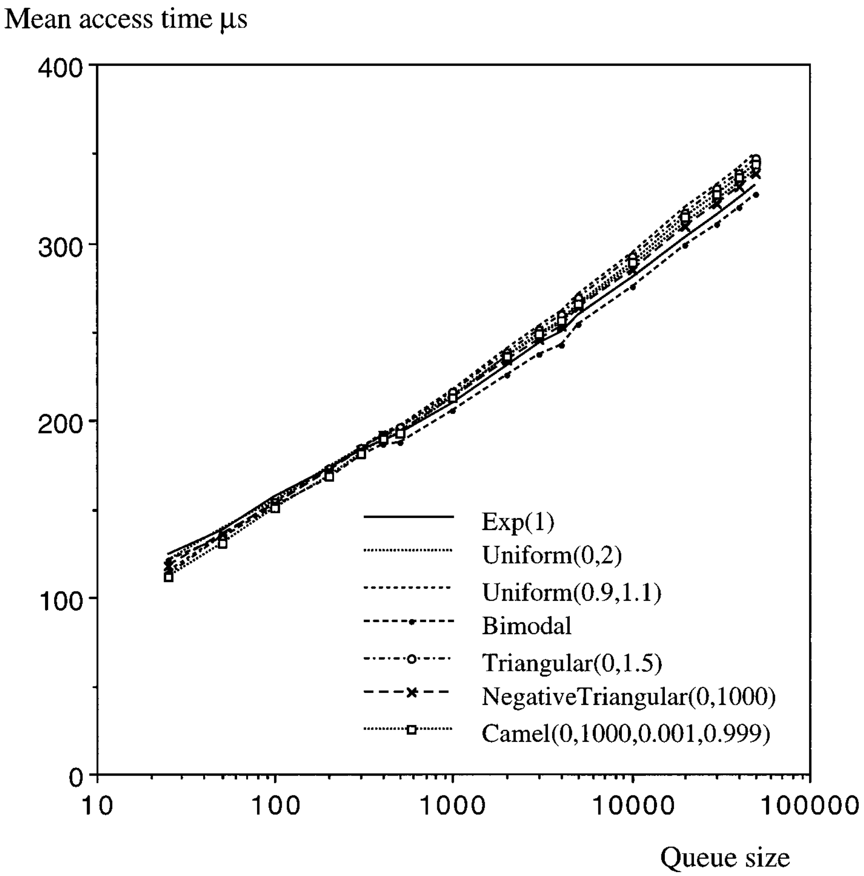


Fig. 2. Mean access time for Binary Heap and Classic Hold experiments.

experiments also show that the Median Pointer Linked list (and linear linked lists in general) is sensitive to the priority increment distribution. It performs best for distributions that are heavily biased, that is, distributions for which new elements tend to fall close to one end of the list. In this case, only a small fraction of the list has to be traversed before finding the correct place to insert the new element in the list.

The access time of the Skew Heap [Sleator and Tarjan 1986], Figures 6 and 7, grows as O(log(n)). However, it consistently shows better performance than the Implicit Binary Heap. The heuristic balancing of the Skew Heap appears to be efficient as it only shows marginal dependency on the priority increment distribution.

The Splay Tree [Sleator and Tarjan 1985] shows the expected O(log(n)) performance, Figures 8 and 9. It outperforms both the Implicit Binary Heap and the Skew Heap but is more sensitive to the priority increment distribution.

In Figure 10, the Calendar Queue [Brown 1988] exhibits a near-linear behavior for most of the distributions in the Classic Hold experiments. In

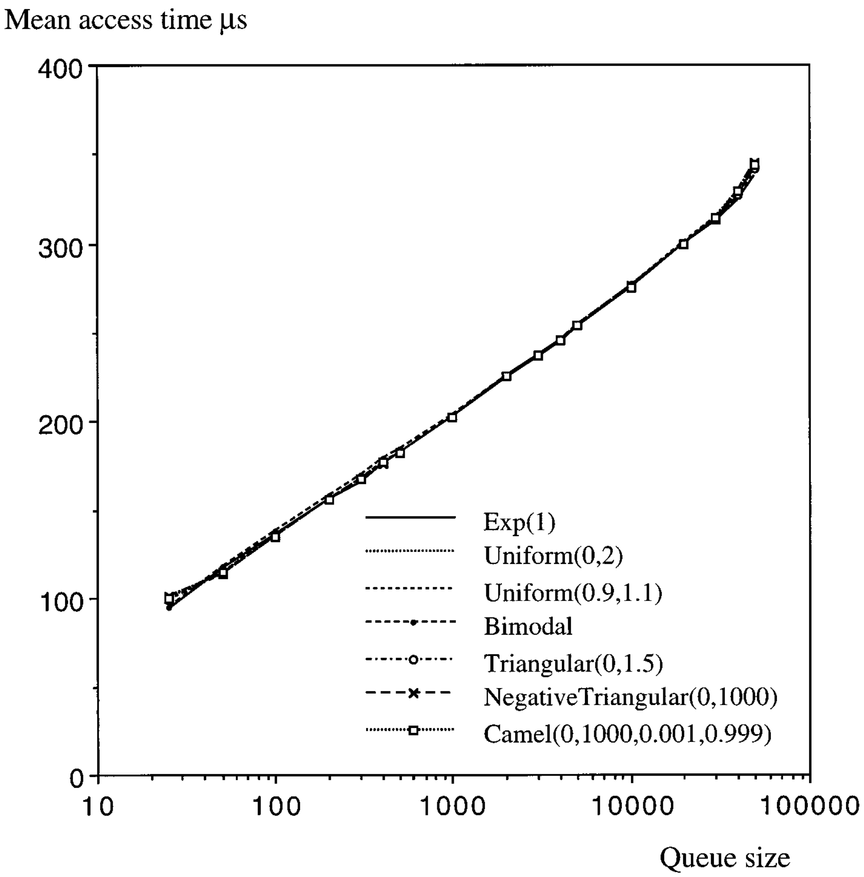


Fig. 3. Mean access time for Binary Heap and Up/Down experiments.

the cases where the performance remains stable, it outperforms the Splay Tree for queue sizes larger than a few thousand elements. However, it also exhibits an irregular behavior for the camel distribution for which the access time grows almost linearly. The camel distribution reveals some of the weaknesses in the Calendar Queue (Figure 10). Its performance is based on the ability to spread the elements evenly over the subintervals (days) creating short sublists. If a large portion of the elements is clustered within a small priority interval and the remaining elements are scattered over a relatively large priority interval, the Calendar Queue is likely to exhibit O(n) performance [Rönngren et al. 1993a]. If the clustered elements are not spread over many subintervals, there will be linked lists of O(n) length. If, on the other hand, the clustered elements are spread out, subsequent scattered elements will fall into different years, forcing a scan of each of the O(n) subintervals in dequeue operations.

Another factor explaining the poor performance of the Calendar Queue for the camel distribution is the heuristics for calculating the length of a subinterval in the resize operations. In the original heuristics suggested by

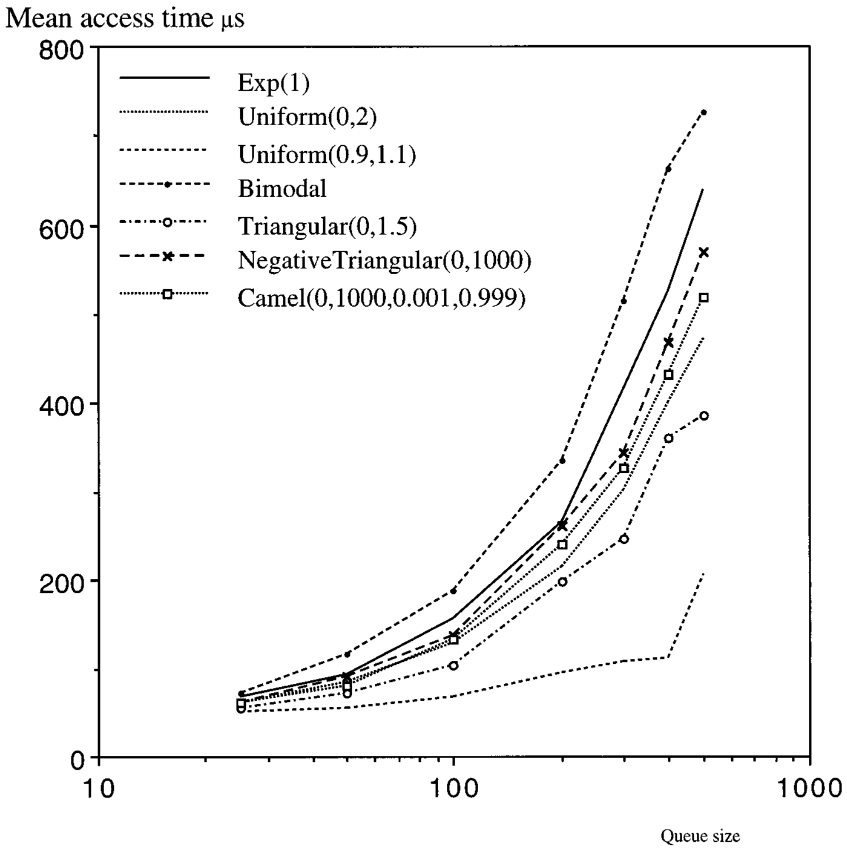


Fig. 4. Mean access time for Median Pointer Linked List and Classic Hold experiments.

Brown [1988], the length of the subintervals is determined by the interpriority distances of the first few hundred elements in the queue. However, this sample may not give a correct picture of the distribution of the elements (priorities) in the queue. Resize operations are only performed when the queue size is halved or doubled. Thus the queue does not adapt itself to dynamic changes in the distribution of the elements unless the queue size changes by more than a factor of two. Other heuristics for the resize criteria have been proposed by Davison [1989] (these have not been implemented in this study). However, the original Calendar Queue has been successfully used in real-world simulations by the authors as well as others [Das et al. 1994].

The Up/Down experiments, Figure 11, reveal that the resize operations are costly, and that the Calendar Queue is not a good choice for applications where the queue size varies often by more than a factor of two.

The Lazy Queue [Rönngren et al. 1993a] shows a near linear performance in the Classic Hold experiments, Figure 12. However, it shows some sensitivity to the priority increment distribution. The humps of the curves coincide with resizes that have been made during the buildup phase. The

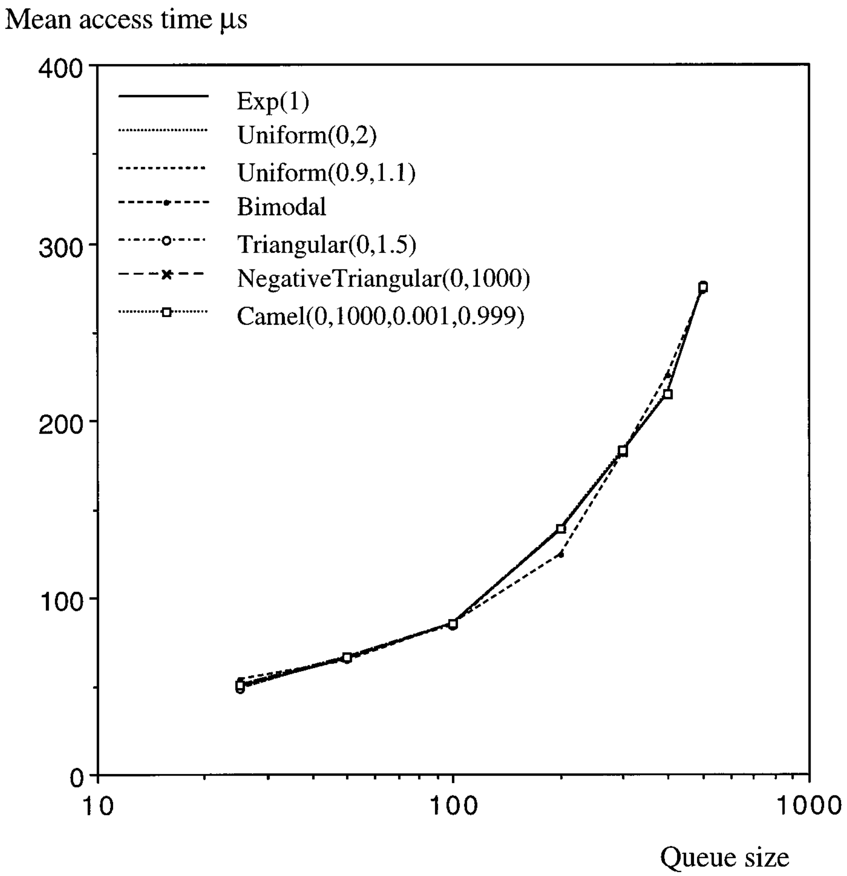


Fig. 5. Mean access time for Median Pointer Linked List and Up/Down experiments.

Up/Down experiments, Figure 13, reveal that the resize operations are costly. The access time remains bounded for all distributions and does not tend to grow with the queue size as is the case for the Calendar Queue, Figure 11. The performance results for the Lazy Queue presented in this study deviate slightly from the results presented in [Rönngren et al. 1993a]. In particular, the steady-state performance is slightly worse, and the Up/Down performance is better in the results here. This can be attributed to the modified resize operations.

Figures 14 and 15 depict the performance of Henriksen's [1977] priority queue used in the GPSS/H simulation system. For queue sizes of up to several thousand elements, it shows a near O(log(n)) behavior. For some distributions, it behaves almost as well as the Splay Tree, Figures 8 and 9. However, for larger queue sizes, it appears to be sensitive to the priority increment distribution.

The Skip List [Pugh 1990] shows a performance that is rather dependent on the priority increment distribution in the Classic Hold experiments,

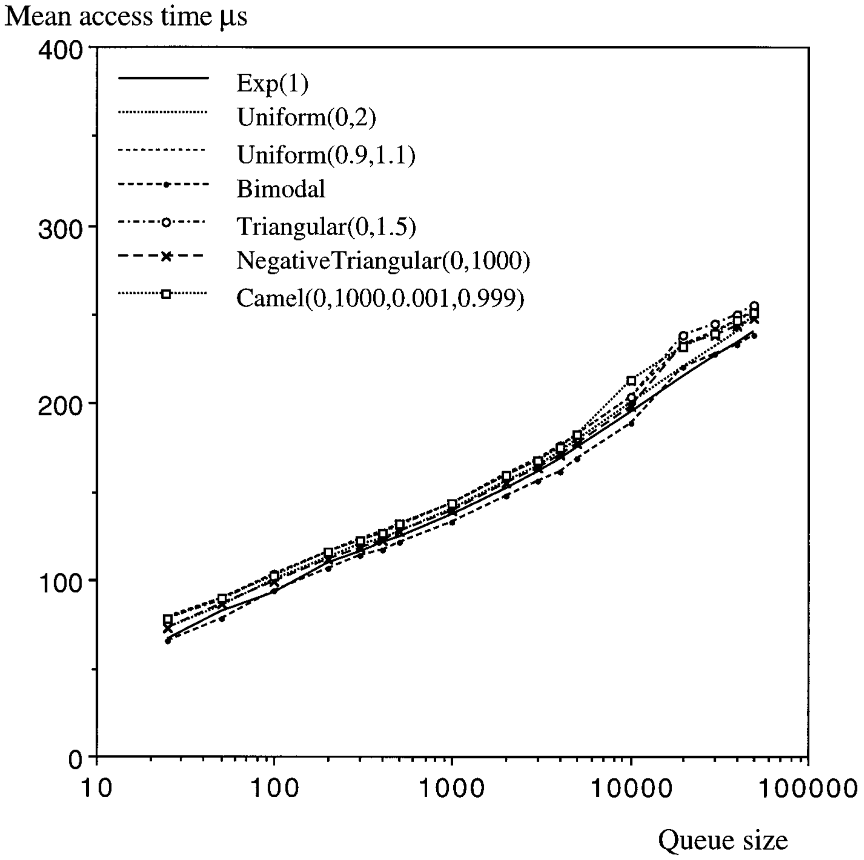


Fig. 6. Mean access time for Skew Heap and Classic Hold experiments.

Figure 16. The access time grows as O(log(n)). The Up/Down experiments, Figure 17, do not reveal any sensitivity to changing queue sizes.

Figures 18 and 19 show the performance of the SPEEDESQ [Steinman 1992]. The SPEEDESQ performs very well for small queue sizes where it can compete with the Median Pointer Linked list, Figures 4 and 5. It is, however, sensitive to the priority increment distribution. Sorting and, in particular, merging the enqueue list with the dequeue list can be time consuming. If the enqueue list contains elements with time-stamps that fall very far into the future, the merge operation has to traverse almost the entire dequeue list, which is an O(n) operation. If such sorts and merges are frequently invoked, the access time will grow linearly as is the case for some of the distributions in the Classic Hold experiments, Figure 18. In the Up/Down experiments, Figure 19, only one sort operation is invoked which explains the better performance than in the steady state.

5.1.2 *Performance on the SUN Sparc10 and the PentiumPro-Based PC.* Figures 20 and 21 depict the performance for all tested priority queues for the exponential priority increment distributions on the SUN Sparc10 and

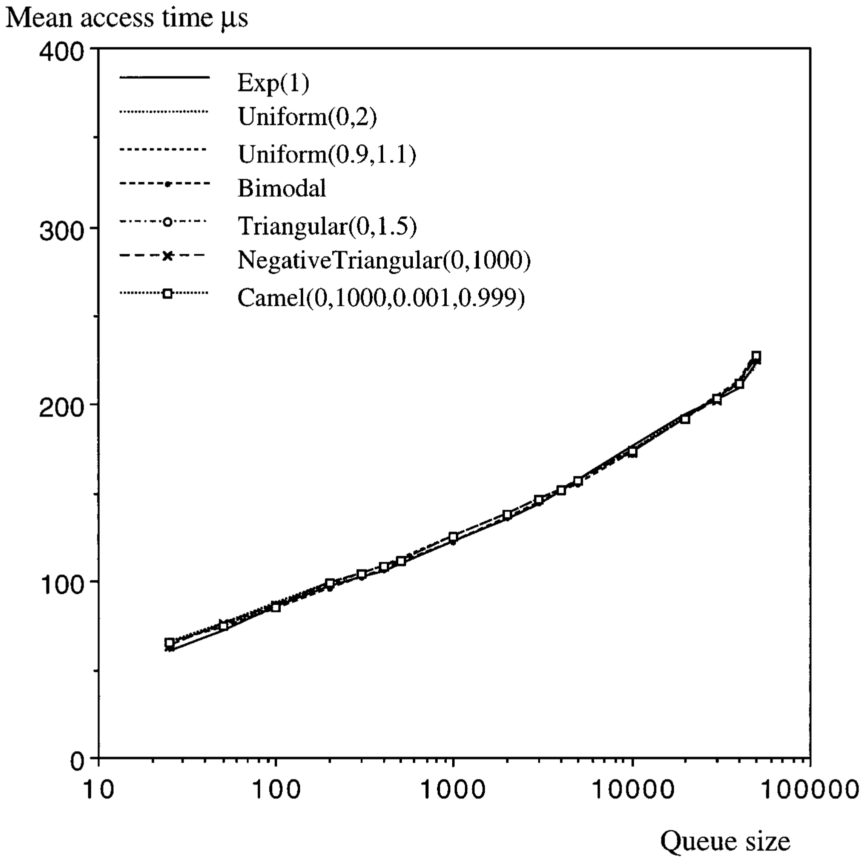


Fig. 7. Mean access time for Skew Heap and Up/Down experiments.

the PentiumPro-based PC, respectively. From these figures, we see that all queues show similar performance on these two architectures. The Lazy Queue and the Calendar Queue also perform well for queue sizes larger than 10,000 events. This is partly due to good cache performance for these queues. It is also interesting to note that the Calendar Queue shows performance comparable to that of the best performing queues down to queue sizes of 50 events. The Splay Tree, the Skew Heap, and Henriksen's algorithm form a second group of curves that show good performance for queue sizes up to approximately 5,000 events. After this point, there is a knee in these curves due to declining cache performance. The relatively poor performance of the SkipList is also notable. This can partly be explained by the fact that our implementation uses the random( ) function supplied in BSD UNIX which, for the Solaris operating system, is time consuming. In Table III, the relative performance for exponential priority increment distribution on the different architectures is found. In our experience, as indicated in Table III, the relative performance of the queues remains similar over the different architectures (with the exception of the SkipList and the Calendar Queue discussed previously). This indicates that

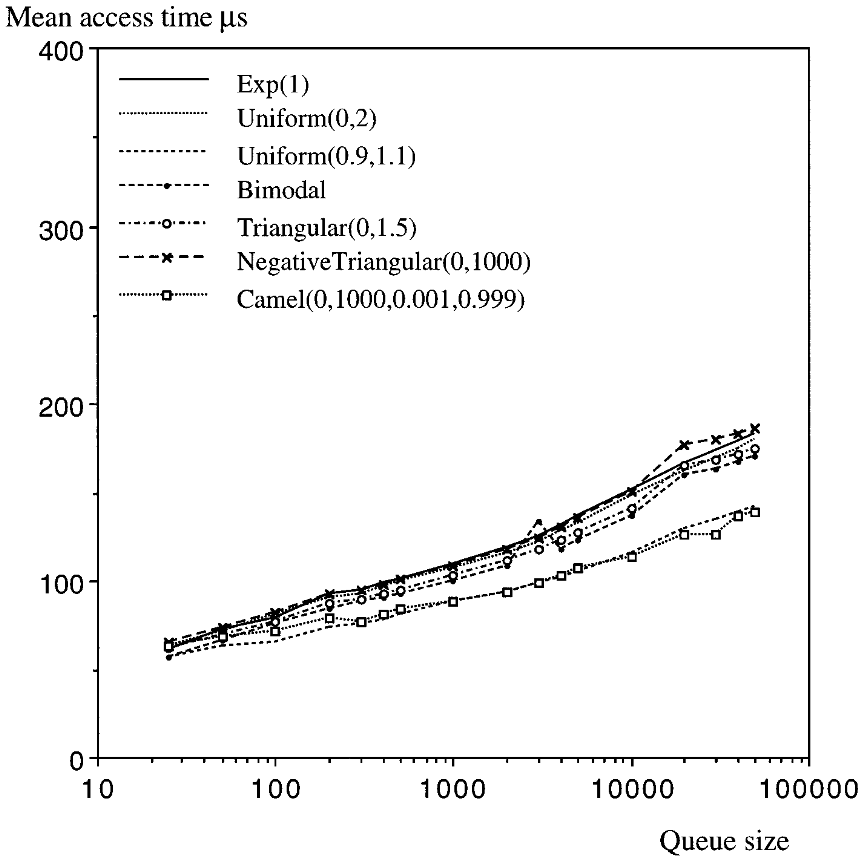


Fig. 8. Mean access time for Splay Tree and Classic Hold experiments.

conclusions from the experiments on the Sequent Symmetry are also relevant for more modern architectures.

Before summarizing the results of the experiments with Classic Hold and Up/Down, we revisit the issue of the impact of bias on the performance of the priority queues. Figures 22 and 23 depict performance results for Bimodal and ExponentialMix distributions with Classic Hold on the SUN Sparc10. When contrasted with the performance results from the Exponential distribution in Figure 20, we can note that only the SPEEDESQ and the SkipList are appreciably sensitive to low bias.

To summarize this section, we note that the Calendar Queue, Splay Tree, and Henriksen's queue, which are all stable, show good performance for queue sizes ranging from 50 to 5,000 elements. The Calendar Queue and the Lazy Queue show good performance for queue sizes larger than 5,000 elements. For very small queues ranging from 25 to 50 events, the Splay Tree, SPEEDESQ, and Skew Heap show good performance with Henriksen's as a strong contender. The Median Pointer Linked list appears to be best suited for queues that do not exceed 25 elements. For the choice of data structure to implement the PES in a general-purpose DES system, the

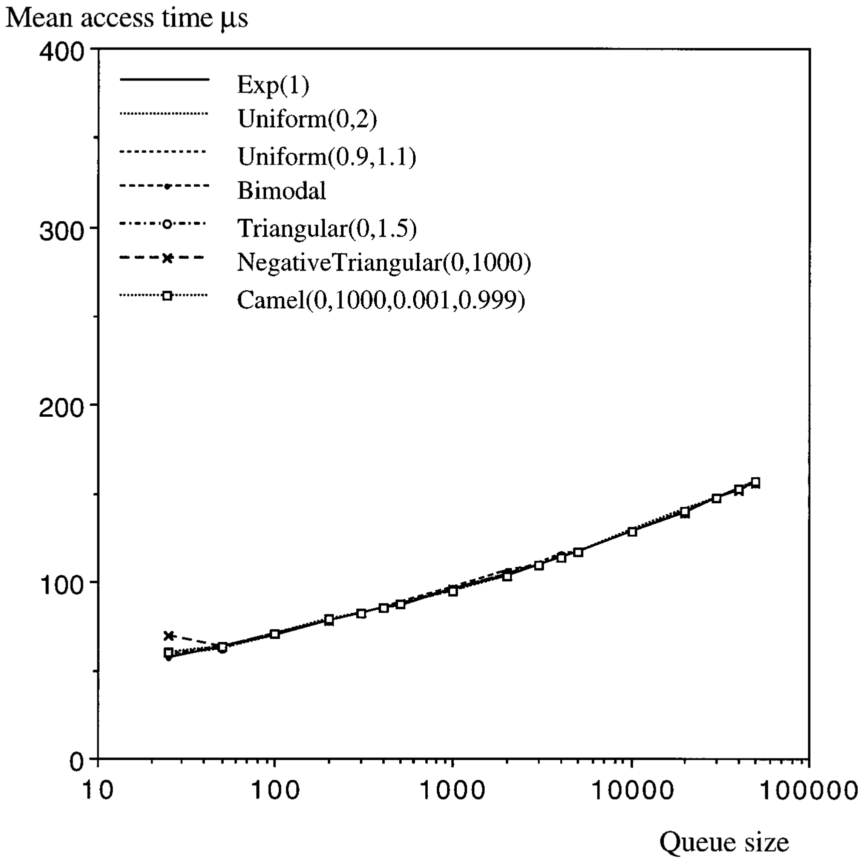


Fig. 9. Mean access time for Splay Tree and Up/Down experiments.

best seems to be either the Calendar Queue, Splay Tree, or Henriksen's queue. It is, however, worthwhile to point out that all these queues have shown some weaknesses which the potential user should bear in mind when making his or her choice. Some of these weaknesses are further explored in the following sections.

5.2 Access Time Limits

The figures presented so far reflect the amortized access times of the queues. For some applications however, it is of interest to identify the maximum or worst-case access time of the queue. One class of applications where this is particularly important is in real-time systems, where such knowledge is necessary to guarantee that the required time constraints can be met. Our measurement method, which measures the time needed for each individual operation on a queue, enabled us to compile such statistics. Most of the previously published works, such as Jones [1986] and Chung et al. [1993], measure only the amortized access time.

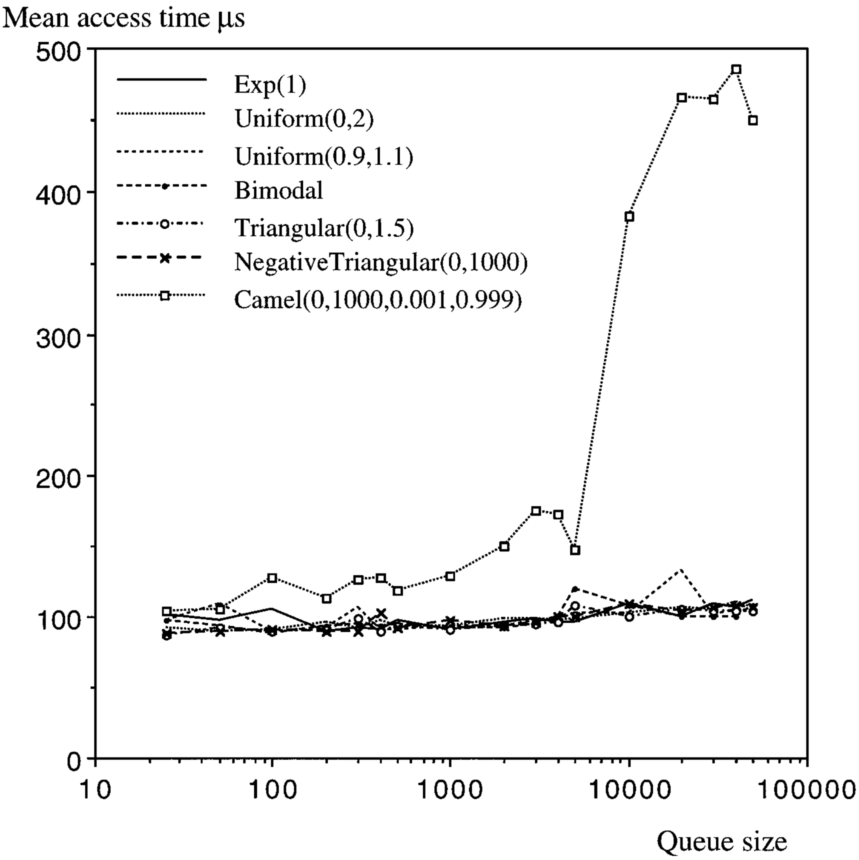


Fig. 10. Mean access time for Calendar Queue and Classic Hold experiments.

In Tables IV and V, the average, maximum, and minimum enqueue and dequeue times, and the corresponding standard deviations for the Classic Hold and Up/Down experiments are presented, respectively. In these experiments, an exponential distribution with mean 1 was used, and the maximum queue size was 1,000 elements. The performance of the queues in these experiments is representative also for other priority increment distributions and queue sizes. In particular, we found that the maximum and minimum access times were virtually independent of the priority increment distribution. Moreover, the queues that fared well in the reported experiments exhibited similar behavior in other experiments that are not discussed in this article. These tables show that the Skew Heap and Splay Tree both have low worst-case access times and low standard deviations. The Implicit Binary Heap shows similar but slightly worse performance than the Splay Tree and Skew Heap. It has, however, a worst-case performance of $O(\log(n))$ for individual operations. Thus the conclusion is that the Splay Tree and Skew Heap are suitable priority queues for real-time applications with soft time constraints where guaranteed worst-case performance is not required, whereas the Implicit Binary Heap is the

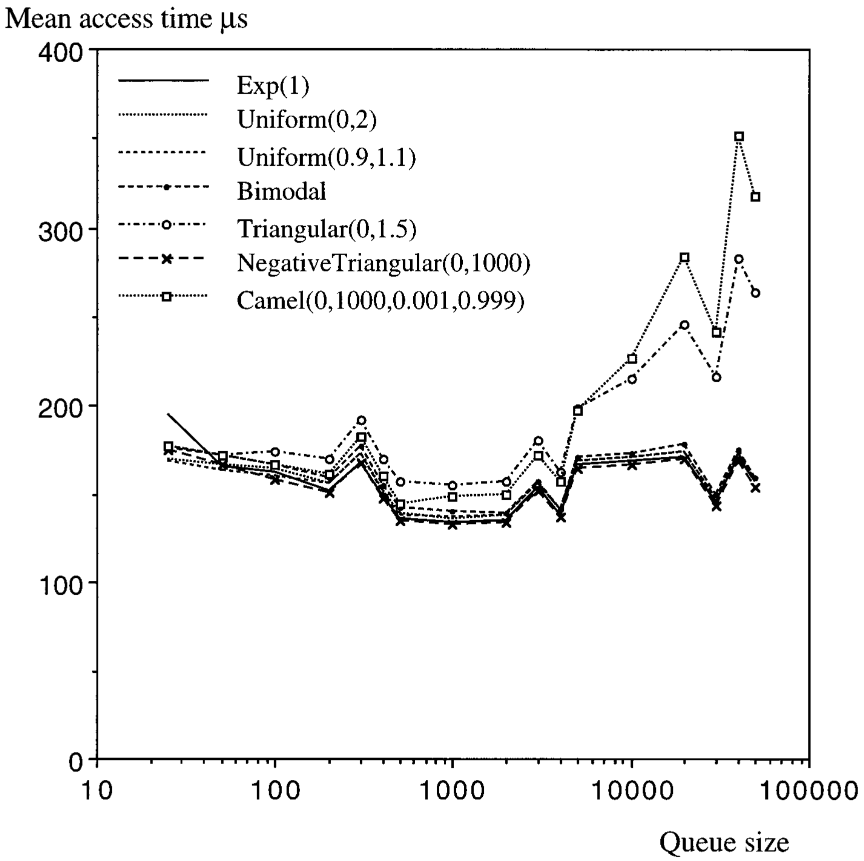


Fig. 11. Mean access time for Calendar Queue and Up/Down experiments.

only choice among the queues in this report that gives guaranteed performance.

Table IV indicates that some dequeue operations on Calendar Queue and Lazy Queue are very time consuming. These operations occur whenever a subinterval is exhausted, and the Calendar Queue needs to search for the next element to dequeue, or when the Lazy Queue must sort a new subinterval. The Up/Down experiment, Table V, also reveals that the resize operations can be extremely time consuming. The Henriksen's queue and Skip List at times have long enqueue times, Tables IV and V. This occurs when the heuristic binary search pointers/tree have to be updated.

5.3 Compound Distributions

The results described in Sections 5.1 and 5.2 are all based on experiments employing single priority increment distributions. However, in a real simulation, several distributions may be mixed and there may appear situations that cannot be covered by using only one distribution. In this section we want to investigate the effects of sudden and drastic changes in

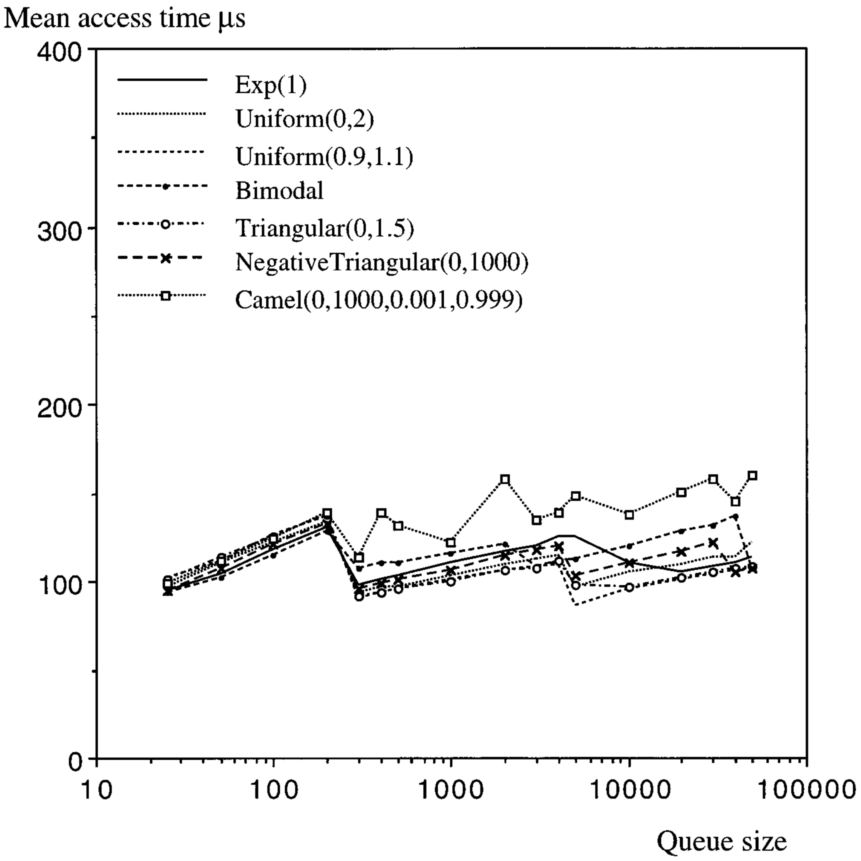


Fig. 12. Mean access time for Lazy Queue and Classic Hold experiments.

bias. Although the Classic Hold experiment usually measures performance in a stable case with regard to bias, the distributions used in this section will cause large variations in bias during the measurement phase. Thus the measurements include what is referred to as transient phases, which occur when bias shifts.

To illustrate when this can occur, we consider a simulation where the activity (i.e., new events) suddenly takes a leap forward in simulated time. This can occur in battlefield simulations where activity is intense during combat and low for long periods between engagements. This behavior can also be found in certain traffic simulations where traffic is heavy during rush hours. In particular we would expect priority queues that are sensitive to bias (or changes in bias) to behave poorly for such models.

To model such phenomena, we used a compound distribution Change (A, B, x) (see also Section 3.2) consisting of sequences of priority increments drawn from two different distributions. In these experiments we chose A, B, and x to stress the priority queues. Two interleaved sequences were used: Change(Triang(90,000, 100,000), Exp(1), 2,000) and Change(Exp(1), Triang(90,000, 100,000), 2,000), where Triang(90,000, 100,000) is a trian-

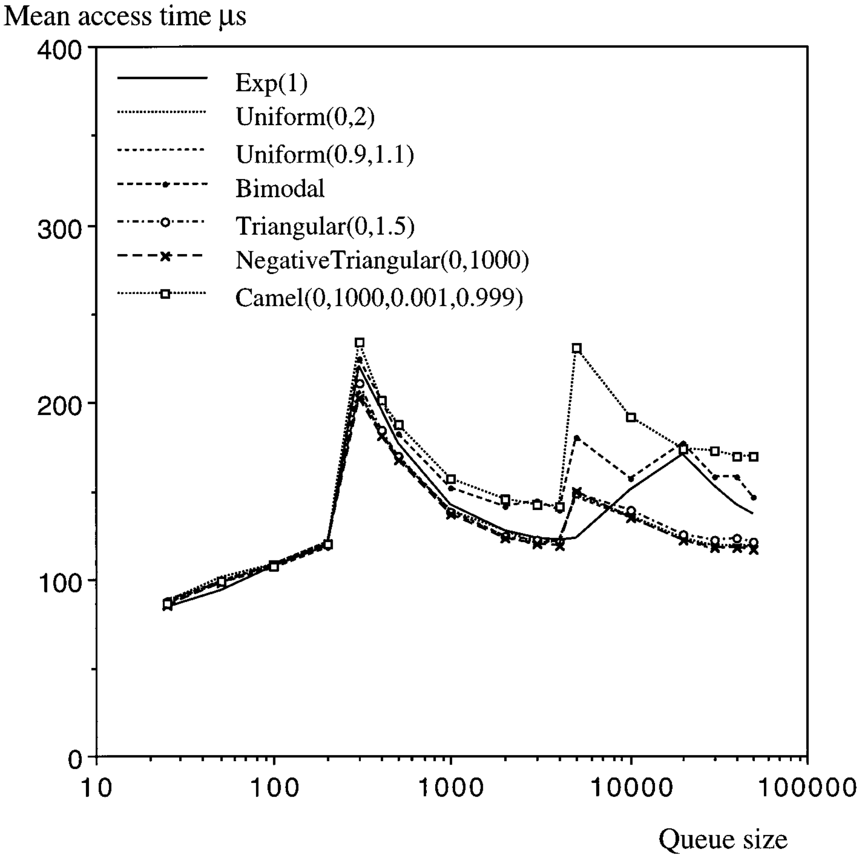


Fig. 13. Mean access time for Lazy Queue and Up/Down experiments.

gular distribution defined on the interval [90,000, 100,000]. The choice of sequences of length 2,000 to be drawn before changing distribution was made to assert that large variations in bias would occur during the measurement phase for queue sizes ranging from 300 to 1,000 events. In these experiments, only one initial value of the random number generator seed was used (as opposed to the other experiments where 10 different seeds were used). In these experiments the bias varies between 0.95 and 0.05 during the time measurement phase. Thus the results from these experiments show the behavior for transient phases with regard to bias. In Figures 24 and 25, the inherent weaknesses associated with the Lazy Queue and the Calendar Queue are clearly depicted. However, the Lazy Queue manages to adapt itself, although at a high cost, but the Calendar Queue heuristics fail. This results in access times of several thousand microseconds.

Inspecting the figures for the Classic Hold experiments, Figures 24 and 25, one observes a transient phase that occurs for most queues with sizes of 300 to a few thousand elements. This is caused by the first change from the initial priority increment distribution to the second distribution and the

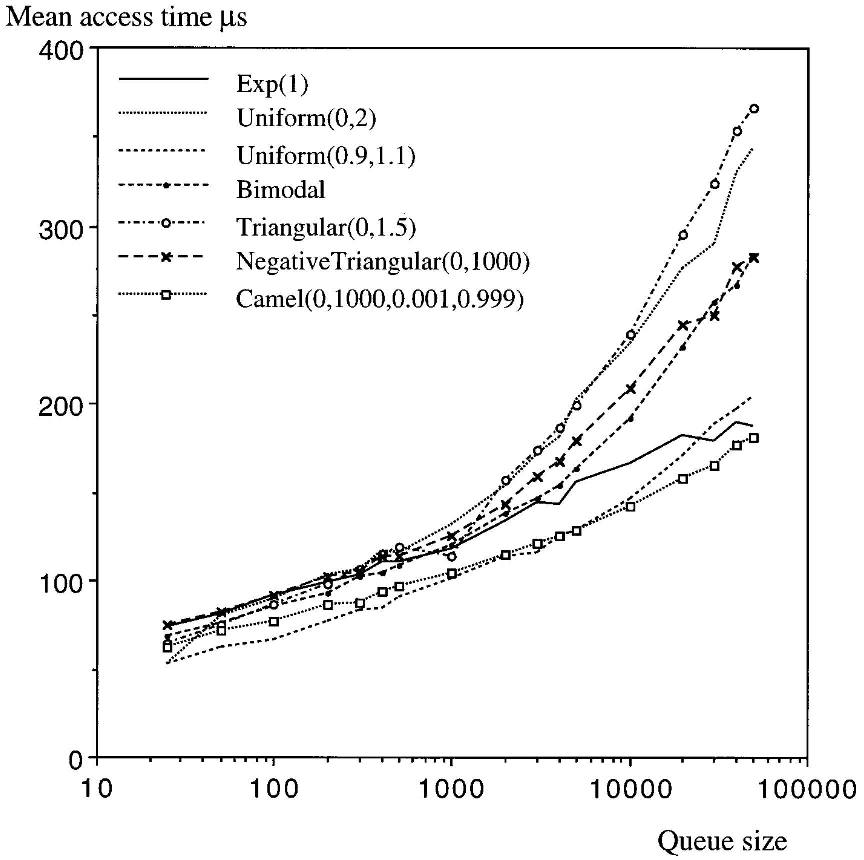


Fig. 14. Mean access time for Henriksen's and Classic Hold experiments.

corresponding shift in bias. The effects of these changes on the priority increment distribution eventually decline as the distribution of the elements in the queue reaches steady state. Interestingly, all priority queues tested are affected by the transient phase. The Calendar Queue fails to adapt itself if the transient phase is not hidden in the buildup phase of the queue, which is the case for queue sizes larger than or equal to 2,000 elements in Figure 25. The dip in the curve for the Calendar Queue in Figure 24 at 5,000 elements is a result of the buildup phase being longer in this case (the results are based on only one experiment for each queue size). In Figure 25, we can also see that the performance of the Skew Heap to some extent can be negatively affected by these kinds of changing distributions. The best performers in these experiments were the Splay Tree and Henriksen's.

5.4 Markov Hold

The Markov Hold model (Section 3.1) proposed by Chung et al. [1993] is a generalization of the Classic Hold model and is supposed to mimic a

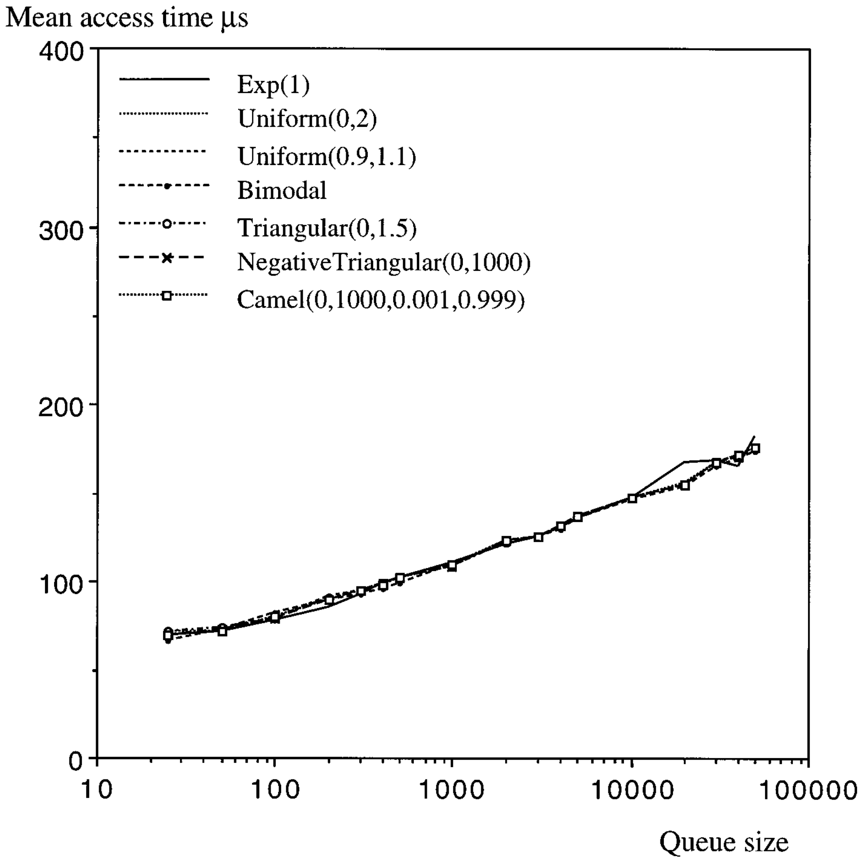


Fig. 15. Mean access time for Henriksen's and Up/Down experiments.

simulation more accurately. The experiments presented in Chung et al. [1993] were run on the same type of computer as the experiments in this study, a Sequent Symmetry. We performed four Markov Hold experiments as described in Chung et al. [1993] allowing us to make direct comparisons of our results with those reported in that paper. In these experiments, the queue initially contained 1,000 elements which were inserted in the random way described for Classic Hold in Section 3.1. A total number of 100,000 operations (enqueue and/or dequeue) were performed on the queues during the measurements. The first experiment, Table VI, corresponds to an uncorrelated random sequence of enqueue and dequeue operations, where each type of operation is equally probable. The second experiment, Table VII, favors insertions with uncorrelated consecutive operations. The third experiment is a Classic Hold experiment, Table VIII. The last experiment shows the performance of an interleaved sequence of enqueue and dequeue operations where each operation has equal probability but with a positive correlation between operations, Table IX. Observe that the rightmost columns in Tables VI through IX show the access times

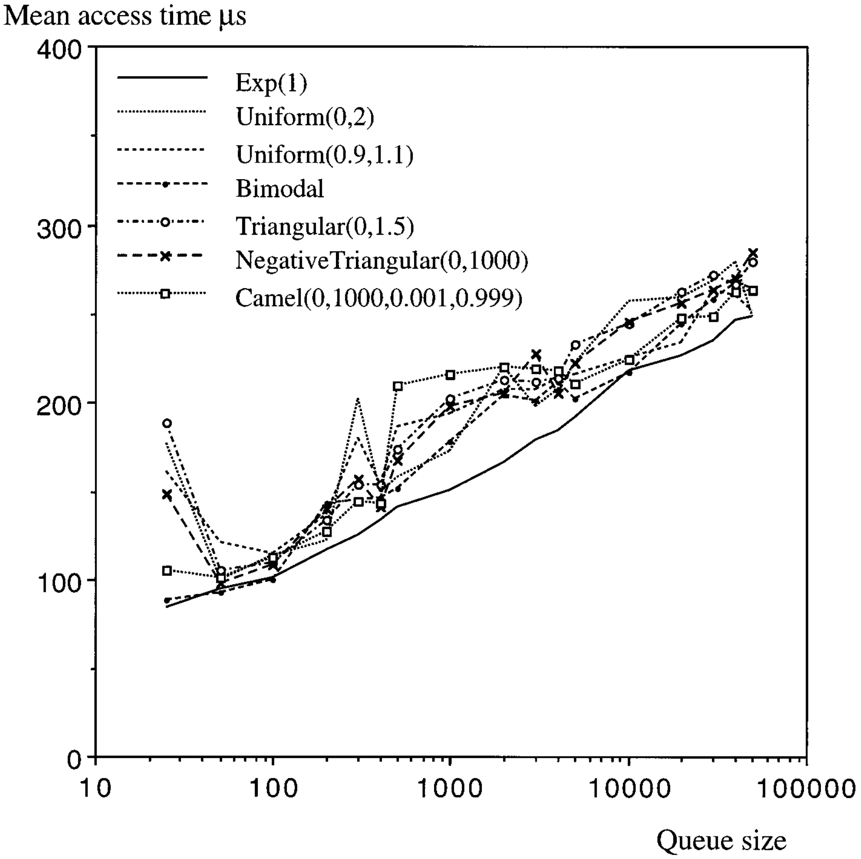


Fig. 16. Mean access time for Skip Lists and Classic Hold experiments.

for the corresponding Classic Hold or Up/Down experiment with an exponential priority increment distribution function.

Our results show that the standard Classic Hold yields results that correspond closely to the Markov Hold experiments with nearly constant queue size, that is, Tables VI, VIII, and IX, columns 2 and 4, respectively. The exceptions are the figures for the SPEEDESQ, where the priority increment distribution plays a distinctive role. For the experiment where insertions are favored, Table VII, the correspondence to the closest Up/Down experiment is low. This can partly be due to the fact that the Markov Hold experiment, in contrast to the Up/Down experiment, did not contain an equal number of insertions and deletions. Most of the queues did not have symmetric enqueue and dequeue times, Tables IV and V.

If we compare the access times to the completion times of an experiment, Tables VII and VIII, we note that the fraction of the time spent in accessing the priority queue is in the range of 20–50% of the completion time. This indicates that when measuring the completion time of an experiment, the performance of the priority queue is obfuscated. On the Sequent Symmetry used in these experiments and in the experiments reported by Chung et al.

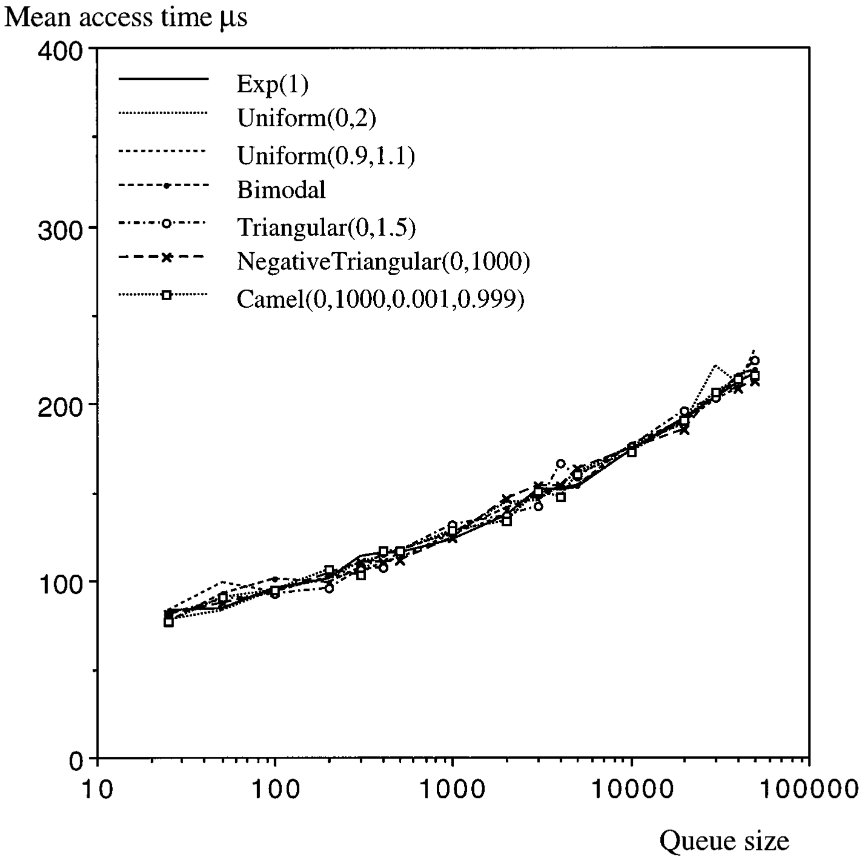


Fig. 17. Mean access time for Skip Lists and Up/Down experiments.

[1993], floating point operations are costly, and a large fraction of the time is spent in the random number calculations. The implementation of the memory management routines (i.e., usage of free lists or direct calls to malloc( )) also has a large impact on the performance. Thus measurement of the completion time rather than the access time may make it harder to evaluate and draw general conclusions from the obtained results.

The Markov Hold model can also be used to perform experiments with pure sequences of either enqueue or dequeue operations. Chung et al. [1993] defined growth (shrink) efficiency as the ratio of the total time to complete $2N$ enqueue (dequeue) operations to the time to complete N enqueue (dequeue) operations on a queue of initial size n. According to these definitions, a growth or shrink efficiency of 2 corresponds to O(1) enqueue and dequeue times, respectively. We, however, choose to define these efficiencies as the ratio of the access times, yielding an efficiency of 1 for a queue with constant time enqueue or dequeue operations. It should be noted that these measures are not independent of n and N, which may make them of limited practical use. Tables X and XI depict the performance for sequences of enqueue (growth efficiency) and dequeue operations

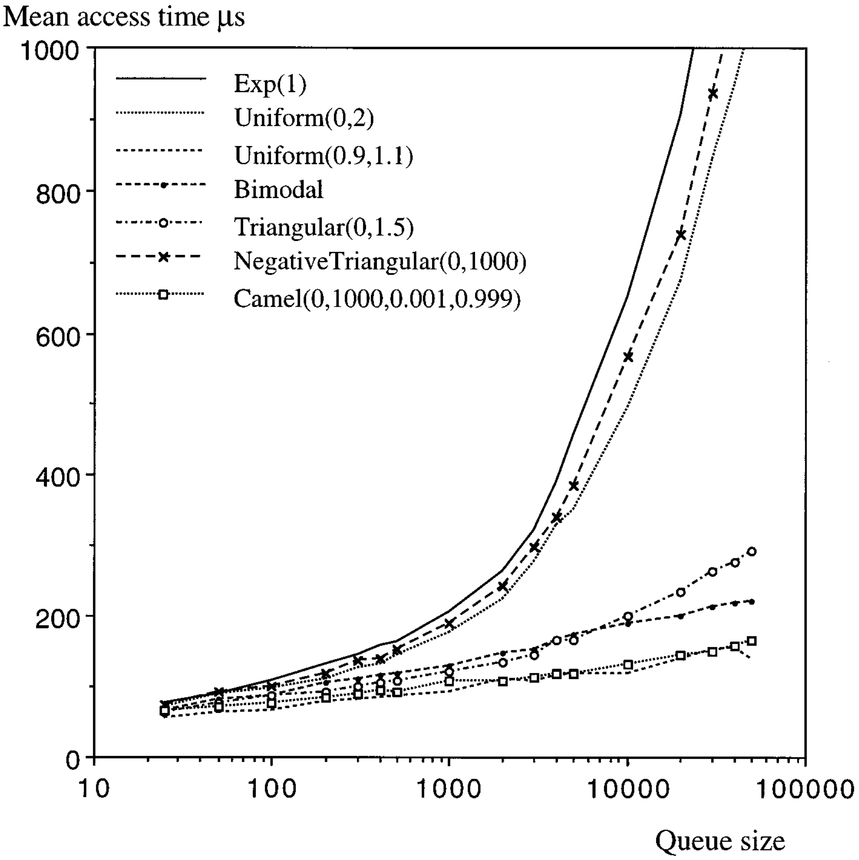


Fig. 18. Mean access time for the SPEEDESQ and Classic Hold experiments.

(shrink efficiency), respectively. The initial queue size n was 8,192 in these experiments and N was 4,096.

These types of experiments may cause misleading results. To illustrate this, we examine the results for the Calendar Queue in Table X. In these experiments, the first enqueue operation invokes a resize operation. The cost of this resize operation is then amortized over 4,096 or 8,192 enqueue operations. This leads to a growth efficiency less than 1, indicating that the access time is reduced as the queue size grows. If, on the other hand, the initial queue size had been 8,193 instead of 8,192, only the last of the series of 8,192 enqueue operations would have led to a resize, resulting in a growth efficiency greater than 1. The same kind of effects can also be noticed for the Lazy Queue.

The SPEEDESQ is another example of how these kinds of experiments can be misleading. In the growth experiments, Table X, the elements are enqueued into an unsorted linked list in constant time. In the shrink experiments, Table XI, at most one sorting and merging of the enqueue list to the dequeue list occurs. This could lead to the conclusion that the SPEEDESQ is not affected at all by changes in the queue size.

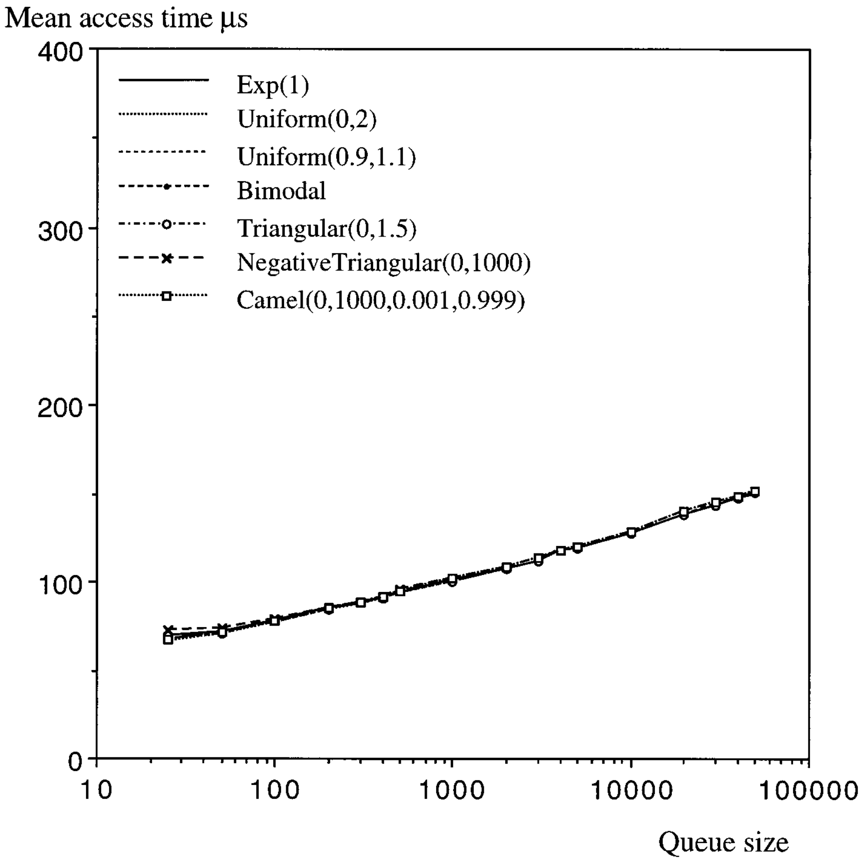


Fig. 19. Mean access time for the SPEEDESQ and Up/Down experiments.

The performance figures in parentheses in Tables X and XI are the corresponding figures from Chung et al. [1993]. The figures found in the columns for enqueue/dequeue are the total times to complete 4,096 or 8,192 enqueue/dequeue operations in microseconds. These figures are, however, hard to interpret. For instance, one can look at the figures for the Calendar Queue in Table X which seem to indicate that 4,096 enqueue operations could be performed in 12 microseconds. If this were correct, the Intel 386i processor of the Sequent Symmetry would have Gips speed, which is obviously not the case.

5.5 Comparison of the Markov Hold and the Classic Hold Model

The Markov Hold model has been suggested as a generalization of the Classic Hold model. It allows random access patterns that could better mimic the behavior of real simulations. It has been claimed that this capability could reveal more information on the performance of priority queues than conventional methods. However, when comparing the results obtained using the Classic Hold and Up/Down with the results obtained

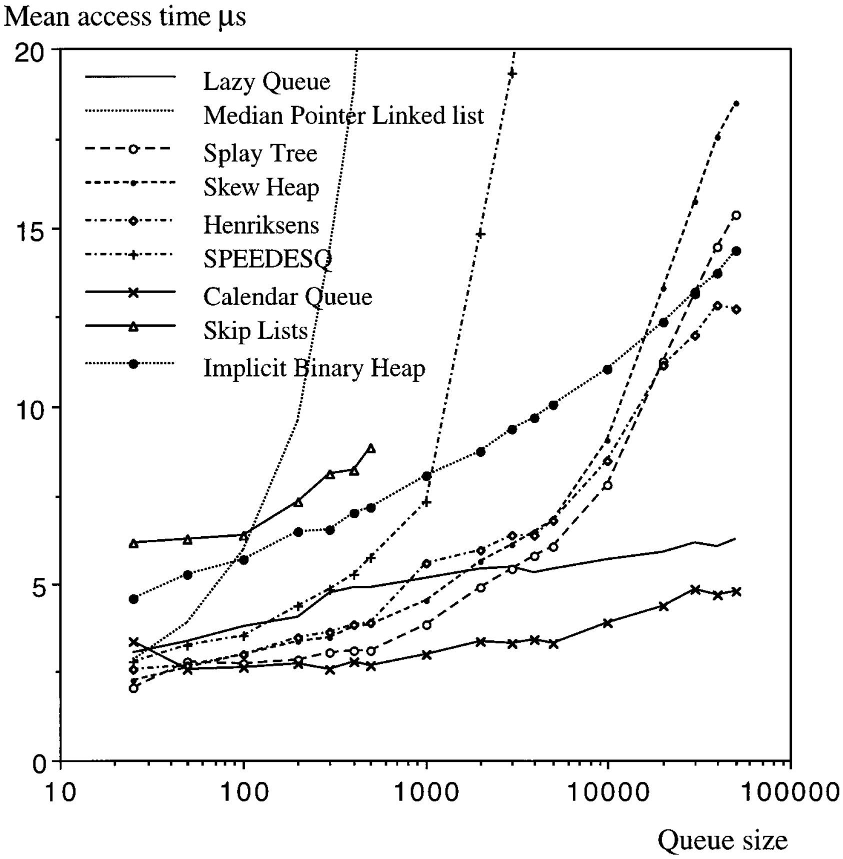


Fig. 20. Access time for Classic Hold experiments and exponential distribution on SUN Sparc10.

from the Markov Hold experiments, we draw the following conclusions. When the queue size remains nearly constant, the Classic Hold model gives as accurate and informative results as the more random access patterns generated by the Markov Hold model for the queues tested in this article. For changing queue sizes, the simple Up/Down access pattern often gives sufficient information. We argue that for these kinds of experiments, the simplicity of the Classic Hold and the Up/Down is appealing, and that it helps reveal more and clearer information on the dependencies of priority increment distributions and queue sizes on the performance of the queue.

5.6 Performance of Parallel Access Queues

The sequential experiments, Sections 5.1 through 5.4, revealed that the sequential counterparts to the parallel access Implicit Binary Heap, Skew Heap, and Lazy Queue are relatively insensitive to the priority increment distribution. The Median Pointer Linked list is somewhat more sensitive but showed good performance for smaller queue sizes. The Lazy Queue, on

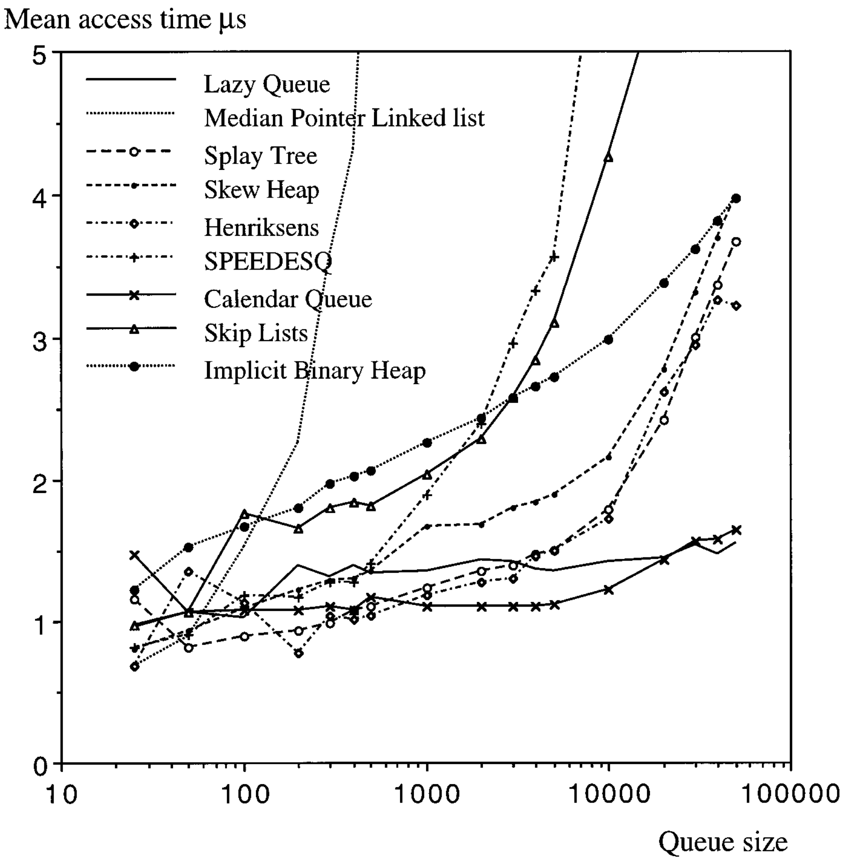


Fig. 21. Access time for Classic Hold experiments and exponential distribution on Pentium-Pro based PC.

the other hand, performed relatively better for larger queue sizes. This allows us to restrict the parallel access experiments to a single distribution, a uniform distribution on the interval [0, 1,000] with bias 0.66. Likewise, we can conclude that only the smaller queue sizes are interesting for the linked list, whereas it is the larger queue sizes that are of interest for the Lazy Queue.

Performance of the parallel access Implicit Binary Heap and linked list in steady-state experiments (i.e., a parallelization of Classic Hold) for queue sizes ranging from 50 to 500 elements are shown in Figures 26 and 27. These queues show relatively good speedup. However, in absolute terms, the performance is not very impressive compared to the best sequential access priority queue implementations; see Section 5.1. Figure 28 depicts the performance for the parallel access Skew Heap (note the logarithmic scale for queue sizes). It also shows good speedup and good performance when compared to sequential priority queues. Figure 29 shows the performance of the parallel access Lazy Queue for queue sizes ranging from 1,000 to 32,000 elements.

Table III. Relative Performance for Exponential Priority Increment Distribution

Queue size	50			100			500			5000			Total
Architecture	SS	S10	PC	SS	S10	PC	SS	S10	PC	SS	S10	PC	Avg. All
Calendar Queue	1.34	1.00	1.20	1.34	1.00	1.20	1.00	1.00	1.13	1.00	1.00	1.00	1.10
Splay Tree	1.00	1.08	1.00	1.00	1.03	1.00	1.05	1.14	1.07	1.42	1.82	1.34	1.16
Henriksen's	1.11	1.05	1.52	1.16	1.13	1.26	1.14	1.44	1.00	1.61	2.04	1.34	1.32
Skew Heap	1.13	1.01	1.06	1.18	1.14	1.21	1.29	1.42	1.31	1.81	2.03	1.69	1.36
Lazy Queue	1.43	1.31	1.30	1.50	1.43	1.13	1.07	1.80	1.29	1.30	1.63	1.21	1.37
Implicit Binary Heap	1.90	2.03	1.72	1.99	2.16	1.86	1.99	2.65	1.99	2.69	3.00	2.44	2.20
SPEEDESQ	1.23	1.27	1.02	1.37	1.34	1.32	1.68	2.11	1.36	4.72	7.70	3.18	2.35
SkipLists	1.30	3.95	1.30	1.29	4.37	1.96	1.45	4.35	1.75	1.99	6.05	2.77	2.71
Median Pointer Linked List	1.13	1.49	1.09	1.18	2.25	1.70	6.59	8.86	6.05	NA	NA	NA	NA

Figures 28 and 29 suggest that the Lazy Queue performs better than the Skew heap when 1 to 3 processors are used, and the queue size is greater than 5,000 elements. However, if more processors are used, the Skew Heap shows better performance. It is only for small or very large queue sizes and a low number of processors accessing the queue in parallel that other queue implementations are preferable.

The overhead associated with the parallel access linked list implementation may seem high compared to the overhead of the other parallel access queue implementations. The reader should, however, keep in mind that these implementations are based on single and double linked implementations, respectively.

Figures 26 through 29 also show that the mean access time (and the speedup) of all the queue implementations converged rather rapidly. The heap-based data structures theoretically allow a number of simultaneous accesses that are equivalent to the number of levels in the heap, that is, O(log(n). Thus the larger queue sizes should be sufficient for further speedup, up to 14 processors. This phenomenon is mainly due to the fact that all accesses must go through one entry point (e.g., the top element of the heap), and this point could be subject to memory contention. This suggests that only a limited amount of parallelism can be efficiently exploited when centralized priority queues are used.

Figure 30 compares the performance of one of the best sequential data structures, the Splay Tree, to the performance of the best parallel access priority queue, the Skew Heap. As indicated in this figure, the speedup is at most 3. Furthermore, the reader should be aware that sharing and migrating data on the Sequent Symmetry S81 are relatively less expensive than on many more recent shared memory architectures which often have

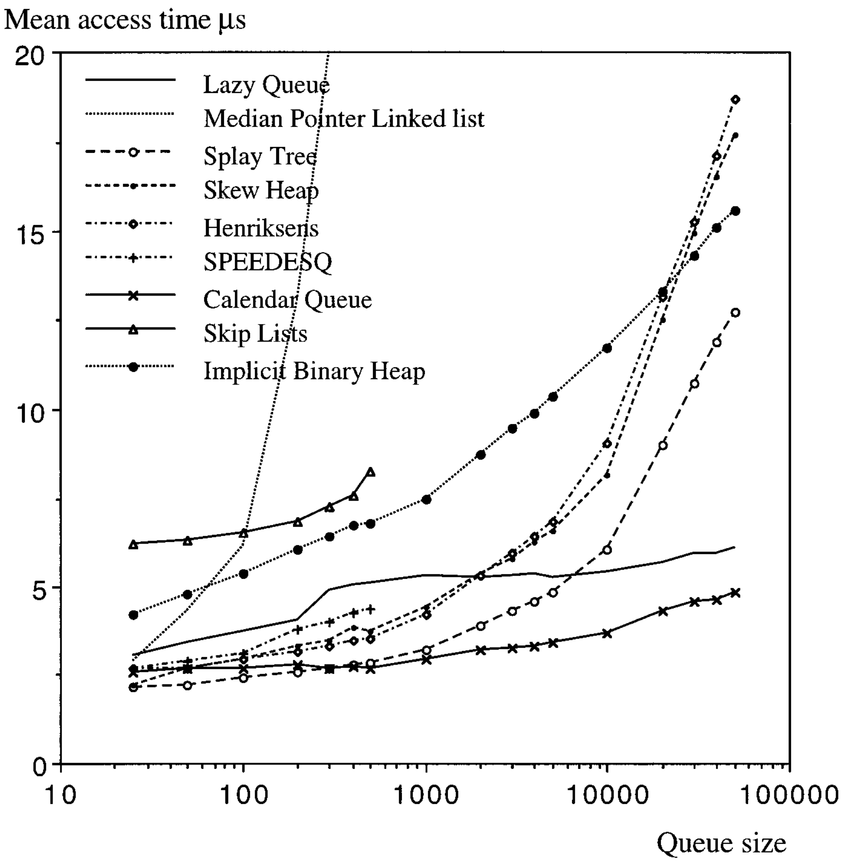


Fig. 22. Access time for Classic Hold experiments and bimodal distribution on SUN Sparc10.

larger data caches allowing for more efficient exploitation of locality and programs with large data sets.

The preceding discussion seems to imply that it is better, from a performance point of view, to partition the data set on several sequential priority queues mapped to different processing elements (PEs) than using a single parallel access queue. Such an approach could allow all PEs to access their respective parts of the data set in parallel. Therefore, this approach could yield optimal linear speedup of the accesses to the data set in the number of PEs used. However, it could also introduce overhead due to the need for additional synchronization or load balancing. The authors have practical experience with situations where the benefits of using a centralized data structure outweigh the drawbacks. An important characteristic of applications using priority queues is the average number of simultaneous accesses to the queue. On a multiprocessor with N processing elements, this number could be expressed as a fraction k of N. In many applications, the accesses to the queue are randomly distributed in time, and k is equivalent to the total fraction of the computation spent in accessing the

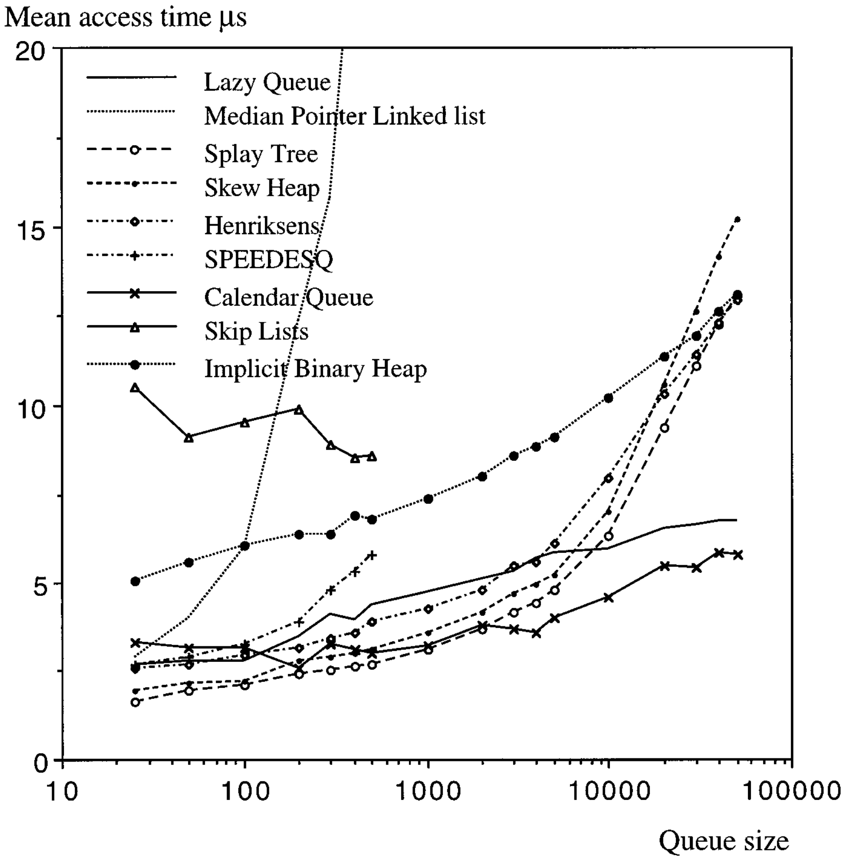


Fig. 23. Access time for Classic Hold experiments and ExponentialMix distribution on SUN Sparc10.

Table IV. Classic Hold, Queue Size 1,000 Elements, Exponential Priority Increment Mean 1

Queue	Enqueue Avg µs	Enqueue Min µs	Enqueue Max µs	Enqueue Std Dev µs	Dequeue Avg µs	Dequeue Min µs	Dequeue Max µs	Dequeue Std Dev µs
Calendar Queue	83	68	282	14	101	91	9090	192
Splay Tree	174	44	411	38	45	31	139	9
Henriksens	200	145	3044	182	37	36	123	4
Skew Heap	111	46	247	21	164	31	387	32
Lazy Queue	88	57	585	43	119	27	6801	635
Implicit Binary Heap	41	29	290	34	378	212	560	37
SPEEDESQ	43	42	131	5	371	35	19637	2066
Skip Lists	263	112	29695	688	40	34	137	6

Table V. Up and Down, Max. Queue Size 1,000 Elements, Exponential Priority Increment Mean 1

Queue	Enqueue Avg μs	Enqueue Min μs	Enqueue Max μs	Enqueue Std Dev μs	Dequeue Avg μs	Dequeue Min μs	Dequeue Max μs	Dequeue Std Dev μs
Calendar Queue	138	69	28128	1029	131	90	16063	596
Splay Tree	154	46	327	41	38	31	131	8
Henriksens	185	55	2894	141	36	35	119	6
Skew Heap	105	45	254	28	140	31	267	35
Lazy Queue	249	74	77818	3349	104	26	5795	436
Implicit Binary Heap	64	22	244	40	342	45	483	60
SPEEDESQ	46	45	123	4	158	35	122197	3863
Skip Lists	209	74	442	57	38	34	120	7

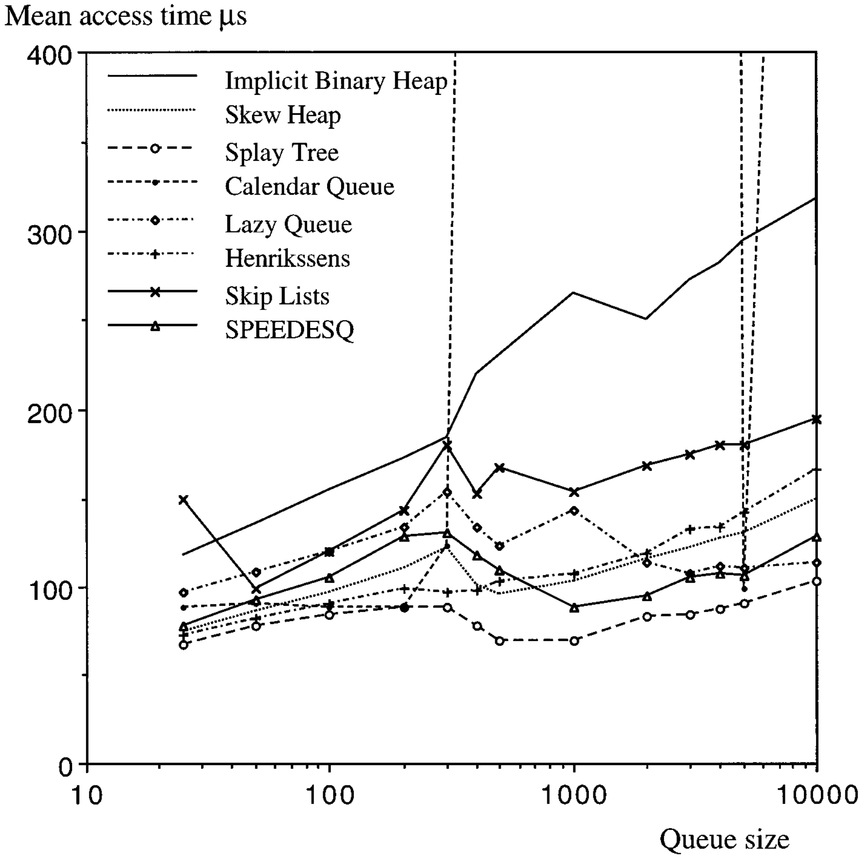


Fig. 24. Access time for Classic Hold experiments, Change(triangular(90,000, 100,000),exp(1),2,000) distribution.

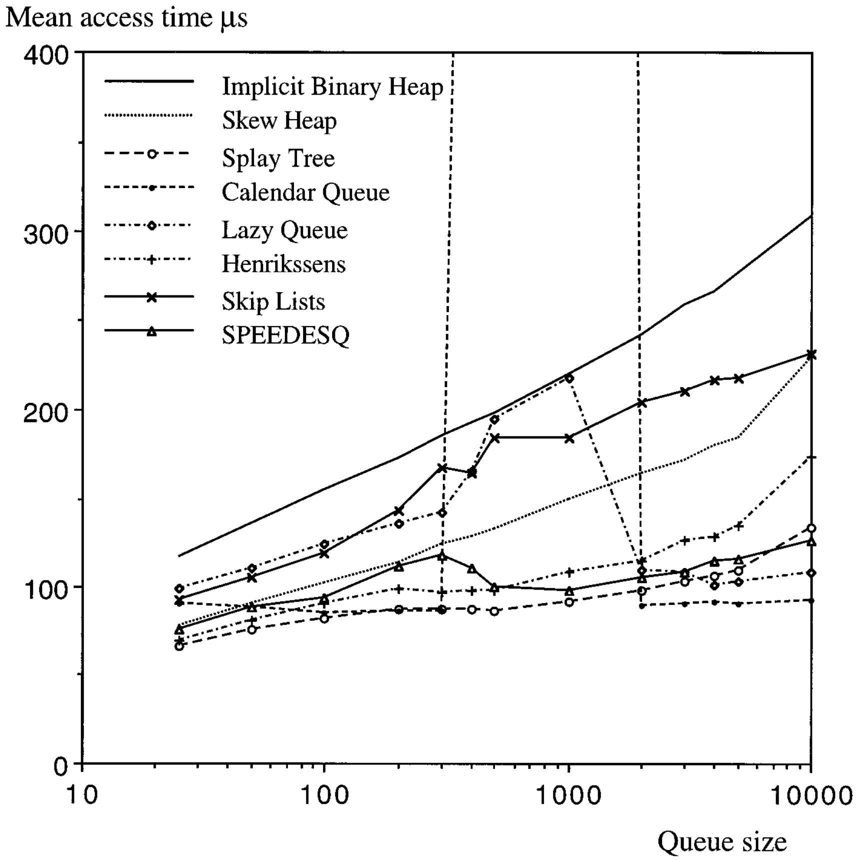


Fig. 25. Access time for Classic Hold experiments, Change(exp(1),triangular(90,000, 100,000), 2,000) distribution.

queue. On the Sequent Symmetry S81 used in the experiments, N was equal to 10. From our experience, k is less than 30% in many applications. That is, the average number of parallel accesses to the priority queue is less than or equal to 3 for which the parallel Skew Heap shows acceptable performance and speedup. Thus the parallel access Skew Heap could be useful in practice. The authors have used the parallel access Skew Heap to successfully implement a centralized scheduling queue for LPs in an optimistically synchronized discrete event simulator [Ahmed et al. 1994]. This opened the possibility to drastically improve the quality of the scheduling which resulted in an overall performance improvement. In particular, it enabled efficient execution of simulation models with a very dynamic nature that could not be efficiently executed using a conventional static partitioning and mapping scheme with decentralized scheduling.

Table VI. Markov Hold, $\alpha = 0.5$, $\beta = 0.5$, Uncorrelated Random Enqueue-Dequeue, and Corresponding Classic Hold

Markov Hold Experiment 1 (Corresponds to Fig. 3 [Chung et al. 1993])	Min Qsize = 646 Max Qsize = 1039		Classic Hold
Queue	Access time μs	Completion time s values in () from [Chung et al. 1993]	Access time μs, Exp(1), Qsize = 1000
Calendar Queue	88	(38)	92
Splay Tree	106	(39)	110
Henriksens	122	(NA)	119
Skew Heap	145	(41)	137
Lazy Queue	105	(NA)	110
Implicit Binary Heap	212	(43)	210
SPEEDESQ	165	(NA)	207
Skip Lists	151	(46)	151

Table VII. Markov Hold, $\alpha = 0.3$, $\beta = 0.7$, Favor Enqueue, and Corresponding Up/Down

Markov Hold Experiment 2 (Corresponds to Fig. 5 [Chung et al. 1993])	Min Qsize = 999 Max Qsize = 41309		Up/Down
Queue	Access time μs	Completion time s values in () from [Chung et al. 1993]	Access time μs, Exp(1), MaxQsize = 40000
Calendar Queue	144	39 (47)	171
Splay Tree	212	46 (60)	153
Henriksens	247	50 (NA)	190
Skew Heap	204	45 (53)	209
Lazy Queue	210	46 (NA)	143
Implicit Binary Heap	235	48 (53)	324
SPEEDESQ	784	NA (NA)	147
Skip Lists	313	55 (75)	212

6. CONCLUSIONS

In this article, we presented the results of an extensive experimental study of several priority queue algorithms. We considered both well-established priority queue algorithms and some recently proposed ones. Three different access patterns (Classic Hold, Markov, and Up/Down) and a variety of distribution functions were used. The access time of each enqueue/dequeue operation was, in many cases, measured separately to isolate the effect of the underlying hardware and to identify the worst-case access time that is of great interest in some real-time systems. We also discussed the perfor-

Table VIII. Markov Hold, $\alpha = 0$, $\beta = 0$ (Hold), and Corresponding Classic Hold

Markov Hold Experiment 3 (Corresponds to Fig. 6 [Chung et al. 1993])	Min Qsize = 999 Max Qsize = 1000		Classic Hold
Queue	Access time μs	Completion time s values in () from [Chung et al. 1993]	Access time μs, Exp(1), Qsize = 1000
Calendar Queue	89	32 (46)	92
Splay Tree	110	34 (45)	110
Henriksens	125	36 (NA)	119
Skew Heap	143	37 (47)	137
Lazy Queue	110	34 (NA)	110
Implicit Binary Heap	182	45 (49)	210
SPEEDESQ	121	NA (NA)	207
Skip Lists	159	38 (52)	151

Table IX. Markov Hold, $\alpha = 0.8$, $\beta = 0.8$, Positive Correlation, and Corresponding Classic Hold

Markov Hold Experiment 4 (Corresponds to Fig. 7 [Chung et al. 1993])	Min Qsize = 461 Max Qsize = 1581		Classic Hold
Queue	Access time μs	Completion time s values in () from [Chung et al. 1993]	Access time μs, Exp(1), Qsize = 1000
Calendar Queue	91	(27)	92
Splay Tree	112	(26)	110
Henriksens	124	(NA)	119
Skew Heap	144	(28)	137
Lazy Queue	110	(NA)	110
Implicit Binary Heap	208	(30)	210
SPEEDESQ	180	(NA)	207
Skip Lists	155	(32)	151

Table X. Markov Hold Growth Efficiency, Initial Queue Size 8192 (Values in parenthesis from Chung et al. [1993])

Queue	Enqueue time μs grow(4096)	Enqueue time μs grow(8192)	Growth efficiency
Calendar Queue	199 (240)	146 (252)	0.73 (1.045)
Splay Tree	245 (427)	252 (902)	1.03 (2.112)
Henriksens	278 (252)	329 (505)	1.2 (2.003)
Skew Heap	158 (351)	154 (514)	0.97 (2.051)
Lazy Queue	92	417	4.5
Implicit Binary Heap	74 (273)	71 (529)	0.96 (1.937)
SPEEDESQ	46	46	1.0
Skip Lists	367 (510)	377 (1033)	1.0 (2.025)

Table XI. Markov Hold Shrink Efficiency, Initial Queue Size 8192 (Values in parenthesis from Chung et al. [1993])

Queue	Enqueue time µs shrink(4096)	Enqueue time µs shrink(8192)	Shrink efficiency
Calendar Queue	101 (53)	138 (860)	1.4 (16.301)
Splay Tree	39 (73)	39 (136)	1.0 (1.867)
Henriksens	42 (38)	42 (76)	1.0 (2.021)
Skew Heap	227 (99)	207 (182)	0.91 (1.838)
Lazy Queue	120	127	1.1
Implicit Binary Heap	510 (313)	470 (579)	0.92 (1.849)
SPEEDESQ	66	54	0.8
Skip Lists	43 (131)	43 (255)	1.0 (1.964)

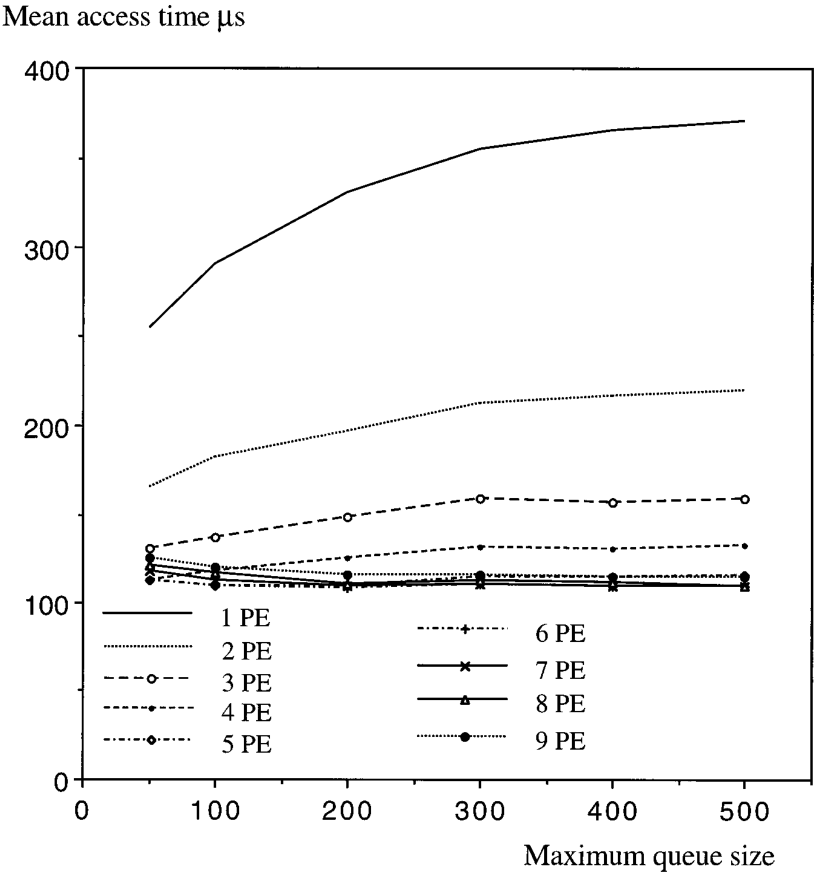


Fig. 26. Mean access time of parallel access Binary Heap in steady-state experiment, uniform(0, 1,000) distribution.

mance of the parallel access priority queues. We tried to identify the impact of a number of factors affecting the performance, such as the number of elements in the queue (queue size), priority increment distributions, and access patterns.

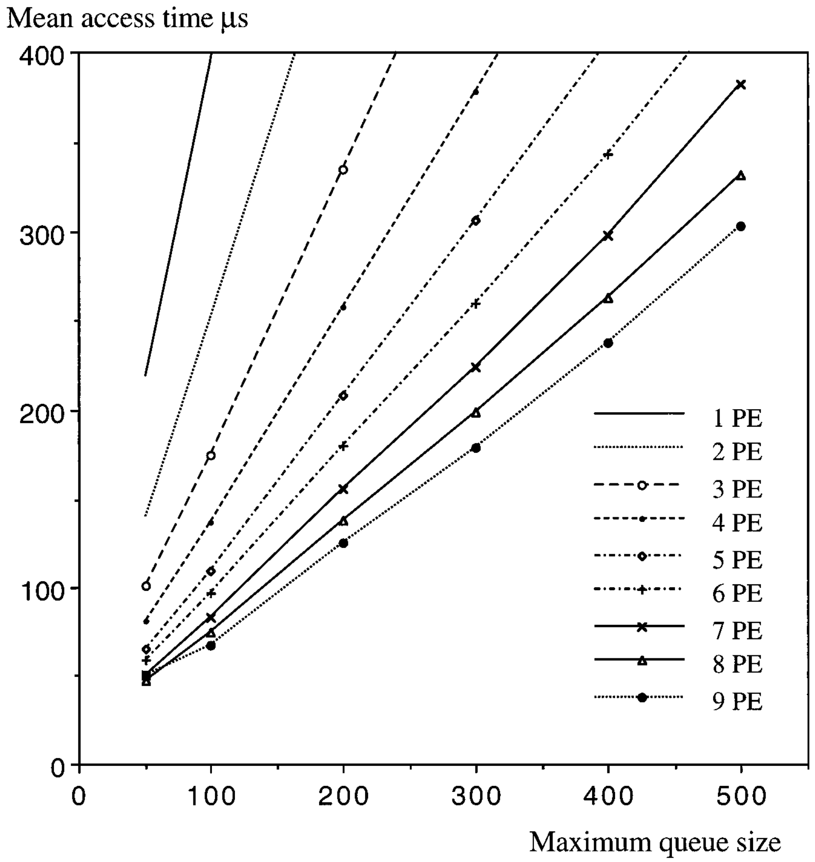


Fig. 27. Mean access time of parallel access linked list in steady-state experiment, uniform(0, 1,000) distribution.

Regarding the access patterns, our conclusion is that it should be kept simple such as the Classic Hold [Vaucher and Duval 1975], Up/Down [Rönngren et al. 1993a], or pure Up and pure Down sequences [Chung et al. 1993]. The main reason for this is that a simple access pattern facilitates the identification of the important factors affecting the performance. Furthermore, the additional knowledge that can be gained by using more complicated access patterns appears to be marginal. However, we believe that the random access patterns that can be generated with the Markov Hold [Chung et al. 1993] model are valuable for further evaluation of priority queues that have been found to perform well for the simpler access patterns.

Our study indicates that performance of some of the recently proposed heuristics-based priority queues, the Calendar Queue [Brown 1988] and the Lazy Queue [Rönngren et al. 1993a], cannot be fully assessed by a single classic priority distribution function. To show the potential weaknesses of these queues, heterogenous distributions or combinations of more than one

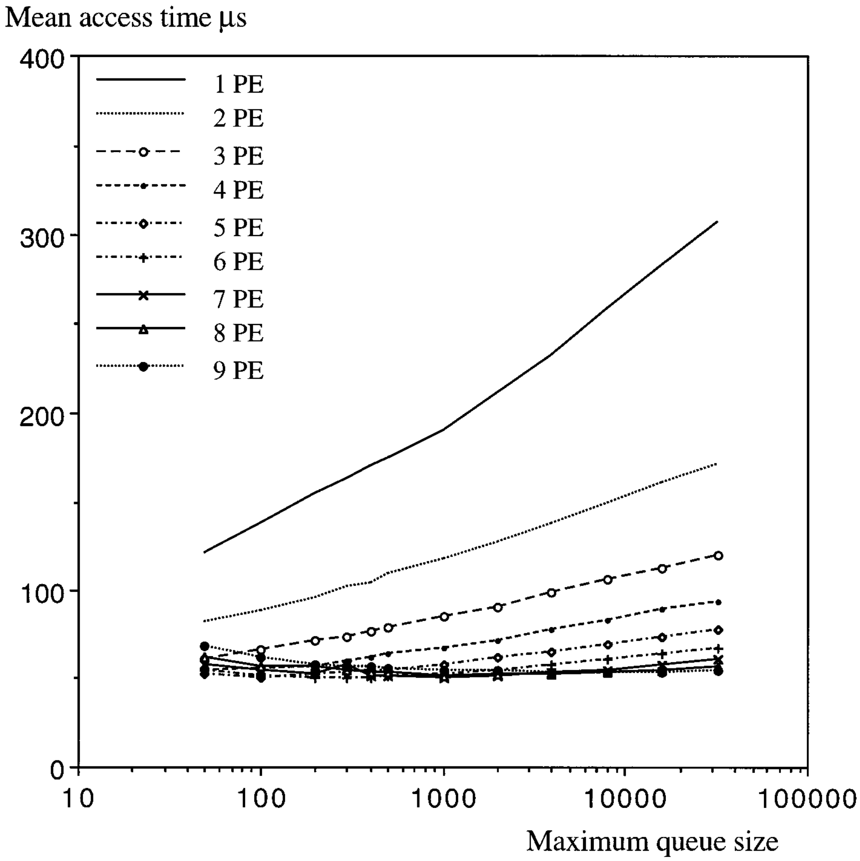


Fig. 28. Mean access time of parallel access Skew heap in steady-state experiment, uniform(0, 1,000) distribution.

distribution have to be employed. Such distributions are helpful in identifying the worst-case access time of some multilist based queues.

The best choice of priority queue algorithm depends heavily on the application. In discrete event simulation, the desirable characteristics are good amortized access time for the range of queue sizes and priority increment distributions used, and in many cases, stability, in terms of preserving FIFO on events with identical priorities (time-stamps). A group of three stable data structures emerged as good candidates for queue sizes ranging from 50 to 5,000 elements which should cover most situations encountered in simulations. This group includes the Calendar Queue, Splay Tree, and Henriksen's queue. The choice is, however, complicated by the fact that all these queues have some weak and strong points. The Calendar Queue performed very well for queue sizes larger than 5,000 elements but it was shown that the heuristics for adaption to the queue size and priority increment distribution can fail in some cases. Henriksen's queue performed well for most operating conditions although it was, in general, somewhat slower than the other two. It was also sensitive to bias in some experi-

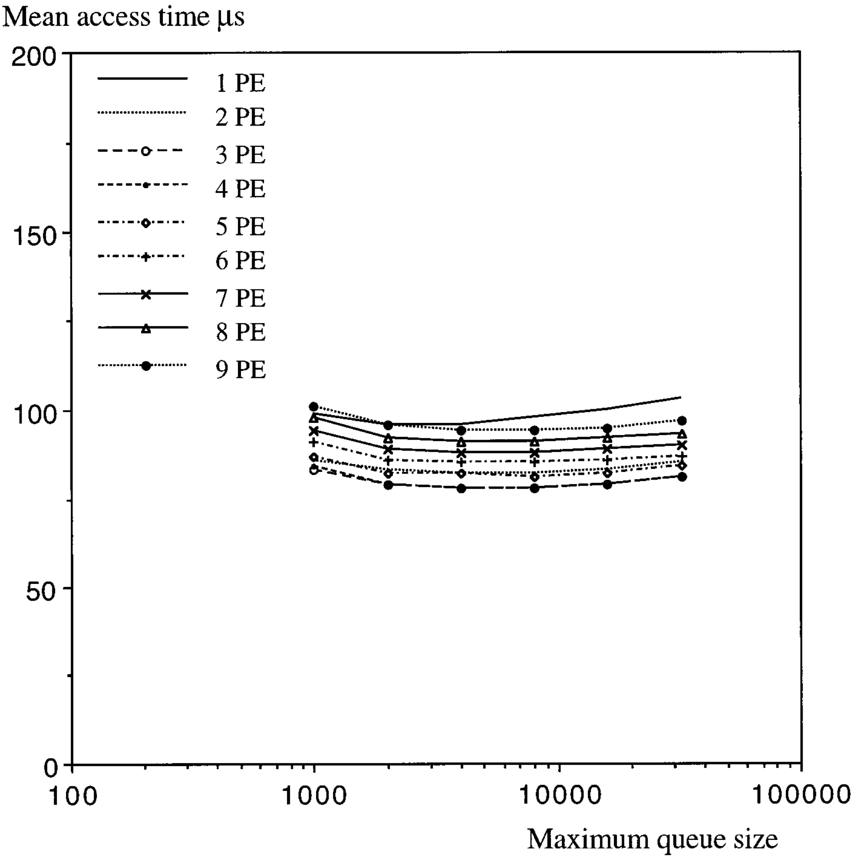


Fig. 29. Mean access time of parallel access Lazy Queue in steady-state experiment, uniform(0, 1,000) distribution.

ments. The Splay Tree often performed well for queue sizes smaller than 50 elements and showed no apparent weaknesses. However, it may not be as simple to implement additional operations on this queue, such as search for and deletion of arbitrary elements, compared to queues based on linked lists, as with the Calendar Queue or Henriksen's. For very small queue sizes (i.e., less than 50 elements), the SPEEDESQ and the Median Pointer Linked list are both reasonable alternatives to the group of three previously mentioned.

For real-time applications, the worst-case access time may be the most interesting measure. Of the tested priority queues, the implicit binary heap had the best worst-case performance for individual operations, which was better than O(n). Thus, if guaranteed performance better than O(n) is required, this is the only choice. The Splay Tree and the Skew Heap, however, showed better amortized worst cases and we did not observe any individual access to these queues with worse time complexity than O(log(n)). Thus they are good alternatives for real-time systems without hard real-time requirements.

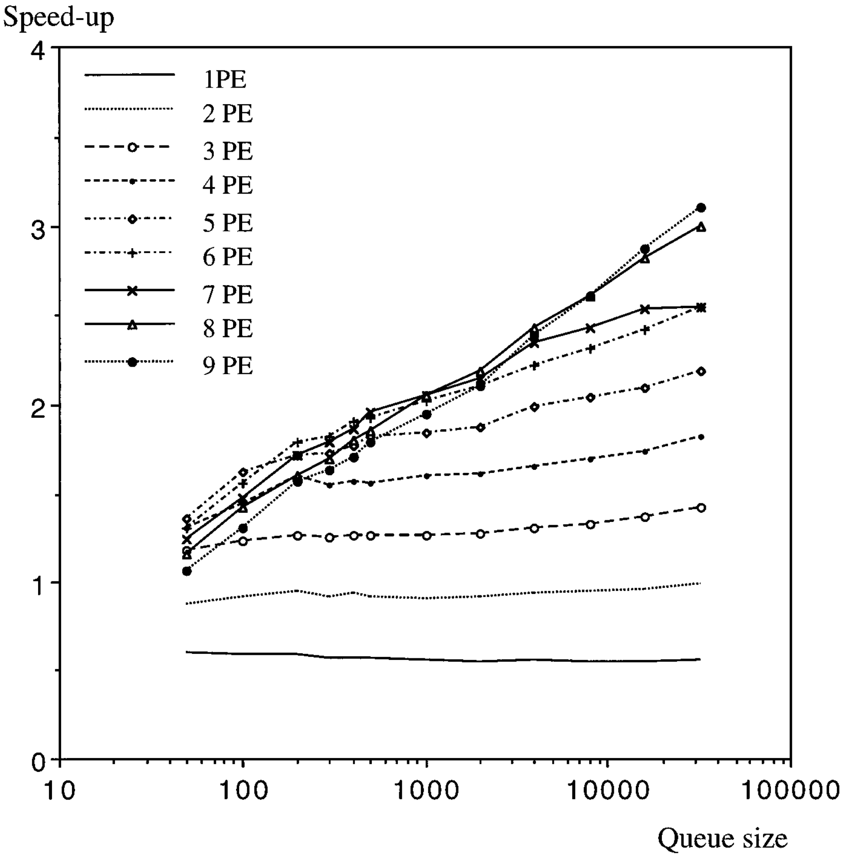


Fig. 30. Speedup, access time of sequential Splay Tree to access time of parallel access Skew Heap for 1 to 9 processing elements.

Several parallel access priority queues were also studied. The commonly used small shared memory computers (such as multiprocessor workstations with typically 8 to 16 CPUs) provide an interesting platform for parallel access priority queues. We observed that in many simulations, the number of concurrent accesses to the PES was less than 30% of the available processors. Our experiments indicate that the parallel access skew heap provides the best performance on small multiprocessors.

ACKNOWLEDGMENTS

We thank the anonymous referees for their insightful comments and suggestions which helped improve this article.

REFERENCES

AHMED, H., RONNGREN, R., AND AYANI, R. 1994. Impact of event scheduling in time warp parallel simulations. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences,* Vol II, 455–463.

AYANI, R. 1990. LR-algorithm: Concurrent operations on priority queues. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Systems,* 22–25.

BAYER, R. AND SCHKOLNIK, M. 1977. Concurrency of operations on B-trees. *Acta Inf. 9,* 1–22.

BENTLEY, J. 1985. Thanks, heaps. *Commun. ACM 28,* 3 (March), 245–250.

BISWAS, B. AND BROWNE, J. C. 1987. Simultaneous update of priority structures. In *Proceedings of the 1987 International Conference on Parallel Processing,* 17–21.

BROWN, R. 1988. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Commun. ACM 31,* 10 (Oct.), 1220–1227.

BLACKSTONE, J. H. HOGG, G. L., AND PHILIPS, D. T. 1981. A two list synchronization procedure for discrete event simulation. *Commun. ACM 24,* 12 (Dec.), 825–829.

CAROTHERS, C. D., FUJIMOTO, R. M., LIN, Y.-B., AND ENGLAND, P. 1994. Distributed simulation of large-scale PCS networks. In *Proceedings of the 1994 MASCOTS Conference.*

CHUNG, K., SANG, J., AND REGO, V. 1993. A performance comparison of event calendar algorithms: An empirical approach. *Softw. Pract. Exper. 23,* 10 (Oct.), 1107–1138.

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

DAVISON, G. A. 1989. Calendar p's and queues. *Commun. ACM 32,* 10 (Oct.), 1241–1243.

DAS, S., FUJIMOTO, R., PANESAR, K., ALLISON, D., AND HYBINETTE, M. 1994. GTW: A time warp system for shared memory multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference,* 1332–1339.

ELLIS, C. S. 1980. Concurrent search and insertion in AVL trees. *IEEE Trans. Comput. C-29,* 9 (Sept.), 811–817.

FUJIMOTO, R. 1990. Parallel discrete event simulation. *Commun. ACM 33,* 10 (Oct.), 31–53.

GORDON, G. 1981. The development of the general purpose simulation system (GPSS). In *History of Programming Languages,* ACM Monograph Series, ACM, New York, 403–426.

HENRIKSEN, J. O. 1977. An improved events list algorithm. In *Proceedings of the 1977 Winter Simulation Conference,* 547–557.

JEFFERSSON, D. 1985. Virtual time. *ACM Trans. Program. Lang. Syst. 7,* 3 (July), 404–425.

JONES, D. W. 1986. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM 29,* 4 (April), 300–311.

JONES, D. W. 1989. Concurrent operations on priority queues. *Commun. ACM 32,* 1 (Jan.), 132–137.

JONES, D. W., HENRIKSEN, J. O., PEGDEN, C. D., SARGENT, R. G., O'KEEFE, R. M., AND UNGER, B. W. 1986. Implementations of time (panel). In *Proceedings of the 1986 Winter Simulation Conference,* 409–416.

KESIDIS, G. AND WALRAND, J. 1993. Quick simulation of ATM buffers with on-off multiclass Markov fluid sources. *ACM Trans. Model. Comput. Simul. 3,* 3 (July), 269–276.

KINGSTON, J. 1985. Analysis of tree algorithms for the simulation event set. *Acta Inf. 22,* 1 (April), 15–33.

KINGSTON, J. 1986. Analysis of Henriksen's algorithm for the simulation event set. *SIAM J. Comput. 15,* 3 (Aug.), 887–902.

LAMPORT, L. 1978. Concurrent operations on priority queues. *Commun. ACM 32,* 1 (Jan.), 132–137.

MCCORMACK, W. M. AND SARGENT, R. G. 1981. Analysis of future event set algorithms for discrete event simulation. *Commun. ACM 24,* 12 (Dec.), 801–812.

MEHL, H. AND HAMMES, S. 1993. Shared variables in distributed simulation. In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation (PADS'93),* 68–75.

MISRA, J. 1986. Distributed discrete-event simulation. *Comput. Surv. 18,* 1 (March), 39–65.

OBAL, W. D. AND SANDERS, W. H. 1994. Importance sampling simulation in UltraSAN. *Simulation 62,* 2 (Feb.), 98–111.

PARK, S. K. AND MILLER, K. W. 1988. Random number generators: Good ones are hard to find. *Commun. ACM 31,* 10, 1192–1201.

PUGH, W. 1990. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM 33,* 6 (June), 668–676.

RAO, V. N. AND KUMAR, V. 1988. Concurrent access of priority queues. *IEEE Trans. Comput. 37,* 12 (Dec.), 1657–1665.

RIBOE, J. 1991. The camel distribution. Tech. Rep. TRITA-TCS-9104, Dept. of Teleinformatics, The Royal Institute of Technology, Stockholm.

RÖNNGREN, R., AYANI, R., FUJIMOTO, R. M., AND DAS, S. R. 1993b. Efficient implementation of event sets in time warp. In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation (PADS'93),* 101–108.

RÖNNGREN, R., RIBOE, J., AND AYANI, R. 1993a. Lazy queue: New approach to implementing the pending event set. *Int. J. Comput. Simul. 3,* 303–332.

SEQUENT 1989. *Guide to Parallel Programming on Sequent Computer Systems.* Prentice-Hall, Englewood Cliffs, NJ.

SLEATOR, D. D. AND TARJAN, R. E. 1985. Self-adjusting binary search trees. *J. ACM 32,* 3 (July), 652–686.

SLEATOR, D. D. AND TARJAN, R. E. 1986. Self-adjusting heaps. *SIAM J. Comput. 15,* 1 (Feb.), 52–69.

STEINMAN, J. S. 1992. SPEEDES: A unified approach to parallel simulation. In *Proceedings of the Sixth Workshop on Parallel and Distributed Simulation,* Vol. 24, No. 3, 75–84.

STEINMAN, J. S. 1994. Discrete-event simulation and the event horizon. In *Proceedings of the Eighth Workshop on Parallel and Distributed Simulation (PADS'94),* 39–49.

TARJAN, R. E. 1985. Amortized computational complexity. *SIAM J. Algebraic Discrete Meth. 6,* 2 (April), 306–318.

UNGER, B. W., GOETZ, D. J., AND MARYKA, S. W. 1994. Simulation of SS7 common channel signaling. *IEEE Commun. Mag. 32,* 3 (March), 52–62.

VAUCHER, J. G. 1977. On the distribution of event times for the notices in a simulation event list. *INFOR 15,* 2 (June), 171–182.

VAUCHER, J. G. AND DUVAL, P. 1975. A comparison of simulation event lists. *Commun. ACM 18,* 4 (June), 223–230.

Received March 1995; revised October 1996; accepted October 1996