Twol-Amalgamated Priority Queues

RICK SIOW MONG GOH and IAN LI-JIN THNG
National University of Singapore

Priority queues are essential function blocks in numerous applications such as discrete event simulations. This paper describes and exemplifies the ease of obtaining high performance priority queues using a two-tier list-based structure. This new implementation, called the *Twol* structure, is amalgamated with three priority queues, namely, the Henriksen's queue, splay tree and skew heap, to enhance the efficiency of these *basal* priority queue structures. Using a model that combines traditional average case and amortized complexity analysis, Twol-amalgamated priority queues that maintain N active events are theoretically proven to offer O(1) *expected amortized complexity* under reasonable assumptions. They are also demonstrated empirically to offer stable near O(1) performance for widely varying priority increment distributions and for queue sizes ranging from 10 to 10 million. Extensive empirical results show that the Twol-amalgamated priority queues consistently outperform those basal structures (i.e., without the Twol structure) with an average speedup of about three to five times on widely different hardware architectures. These results provide testimony that the Twol-amalgamated priority queues are suitable for implementation in sizeable application scenarios such as, but not limited to, large-scale discrete event simulation.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures; F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity; 1.6 [**Computing Methodologies**]: Simulation and Modeling

General Terms: Algorithms, Design, Experimentation, Performance, Theory, Verification

Additional Key Words and Phrases: Algorithm analysis, discrete event simulation, priority queue, future event list, pending event set, calendar queue, Henriksen's, skew heap, splay tree, tree, simulator

1. INTRODUCTION

A *priority queue* is an abstract data type that efficiently finds the highest priority element within the set of elements S contained in the priority queue. The two basic operations are *Insert* and *DeleteMin*, where Insert(e, S) inserts a new element e into the set S so that $S = e \cup S$, and DeleteMin(S) returns the element e with the highest priority (i.e., $e = \min S$) and removes it from the set S so that $S = S - e$.

Authors' address: Department of Electrical and Computer Engineering, National University of Singapore, 3 Engineering Drive 3, Computer Communication Network Laboratory, E4A #05-06, Singapur 117576; email: smgoh@nus.edu.sg.

Priority queues are used in a wide variety of applications. They are used frequently in many problems such as branch and bound algorithms, pattern matching, data compression techniques, operating systems, real-time systems, and discrete event simulations. The metric of interest in these varieties of applications is the time required to perform the most common operations on the embedded priority queue within each application, namely, the Insert and DeleteMin operations. This metric is referred to as *access time* [Rönngren and Ayani 1997]. In real-time systems, the worst-case access time is perhaps the most important criterion in deciding the most suitable candidate out of the numerous proposed priority queue algorithms in the literature. On the other hand, for discrete event simulations, the *amortized* (i.e., average of total cost of executing a sequence of operations on the queue over the number of operations) access time [Tarjan 1985] is undoubtedly the most important criterion in selecting the appropriate priority queue algorithm. This is because from a user's point of view, the total run-time of the simulation job is by far more important than the time taken by each individual priority queue operation.

The assortment of proposed priority queues falls mainly into two categories: tree based and list based. Examples of popular tree-based priority queues are the implicit binary heap [Bentley 1985], splay tree [Sleator and Tarjan 1985], and skew heap [Sleator and Tarjan 1986]. The implicit binary heap is well known for its optimal bound of $O(\log(n))$ in worst-case access time, whereas the skew heap and the splay tree are famous for their $O(\log(n))$ amortized access time. Although these priority queues are endowed with a similar bound of $O(\log(n))$, Rönngren and Ayani [1997] have empirically shown that the skew heap and the splay tree outperform the implicit binary heap in almost all benchmark scenarios. Examples of applications of the splay tree include in the CelKit (formerly known as SimKit) simulator [Gomes et al. 1995], data structure for fast IP lookups [Narlikar and Zane 2001], and is also used in the block-sorting process of Burrows and Wheeler [Yugo et al. 2002].

List-based priority queues are mainly popularized by the widely used calendar queue (CQ) [Brown 1988]. Although the CQ has a worst-case bound of $O(n)$, that is, equivalent to the bound of a linear linked list, the CQ has been employed in various simulation systems such as GTW [Das et al. 1994], CSIM18 [Schwetman 1996], Network Simulator v2 [Fall and Varadhan 2002], as well as in a quality of service algorithm where it maintains real-time packet requests [Stoica et al. 2000]. The reason for its high regard is that the performance of the CQ has been empirically demonstrated to exhibit constant cost per operation frequently under many operating conditions. An interesting note to highlight is that there has also been development of a CQ-like structure as part of a rate controller for ATM switches [Hagai and Patt-Shamir 2001]. Another popular list-based priority queue is the Henriksen's queue [1977] which frequently exhibits $O(\log(n))$ behavior and is bounded by $O(\sqrt{n})$ in amortized complexity [Kingston 1986]. Henriksen's queue is employed in simulators such as SLAM [Pritsker and Pegden 1979], GPSS/H [Henriksen and Crain 1996], and SLX [Henriksen 1997].

An example of the use of priority queues is in the area of discrete event simulation. In a discrete event simulation, a priority queue is used to hold the *future event list* (FEL); the elements in the priority queue are often referred to as *events*. The FEL is defined as the set of all events generated during a simulation that have not been simulated or evaluated yet. The FEL controls the flow of simulation of events with the minimum timestamp having the highest priority and maximum timestamp having the least priority. These events should be processed in nondecreasing time-order with multiple events of equal timestamp being processed in the order that they are inserted into the FEL. If processed in this manner, causality relations between these events would not be violated, ensuring the simulation results to be correct and deterministic. Deterministic simulation results mean that exact duplicate results will be obtained when any two or more simulation runs with identical parameter settings are specified.

Comfort [1984] has revealed that up to 40% of the computational effort in a simulation may be devoted to the management of the FEL alone. In the management of the FEL, the Insert and DeleteMin operations (or sometimes labeled as *enqueue* and *dequeue* operations, respectively, in simulation software context) account for as much as 98% of all operations on the FEL [Comfort 1984]. Understandably, as a simulation system becomes larger, the length of simulation time may take days or weeks to yield results with an acceptable level of statistical error. Hence, the priority queue employed should be efficient especially for large-scale simulations that involve large number of events during the execution of simulation models.

In some cases, the problem of large number of events generated can be minimized using efficient simulation techniques that simplify the simulation model or that which manipulate the statistical properties of the model to reduce the FEL's size. An example is the use of *importance sampling* if the metric of concern is the small cell-loss ratio (e.g., less than 10^{-8}) in an ATM switch [L'Ecuyer and Champoux 2001]. However, such techniques are specific only to certain models of concern and cannot be generalized for most simulations. In addition, most users are only concerned with the simulation models that they have created and are not conscious of the underlying structure of the simulator. Therefore, a good implementation of the FEL should be able to efficiently handle a widely varying FEL size corresponding to small- or large-scale discrete event simulation, without any user's intervention on the simulation engine.

This paper proposes a novel two-tier list-based structure, which we refer to as the *Twol* structure that upon amalgamation with a *basal* priority queue, such as a tree-based structure or the Henriksen's queue, produces very efficient priority queue structures. The reason for using a multiple tier list-based structure as the fundamental structure for Twol is that a list-based structure such as a CQ offers constant amortized performance except for scenarios in which the number of events vary too quickly with time or if the event distribution is skewed. To address this shortcoming, the Twol algorithm employs a two-tier structure with dynamic construction of its operating parameters without any

user's intervention. In addition, a simple theoretical proof is provided to demonstrate that a priority queue functioning as the basal structure amalgamated with the Twol algorithm has O(1) *expected amortized* performance under reasonable assumptions, using a model which combines traditional average case and amortized complexity analysis (see Section 3).

Also, we present a wide range of empirical results obtained from several experimental studies to illustrate that the Twol-amalgamated priority queues perform better compared to the basal queues and that they exhibit near O(1) amortized complexity for widely varying priority increment distributions, as well as for queue size ranging from as small as 10 to as large as 10 million. In addition, these new priority queues are implemented in a C++ process-oriented network simulator called *Swan* (simulator without a name) [Thng and Goh 2004] to provide a realistic testimony that the performance of the Twol-amalgamated priority queues extends beyond the theoretical and static benchmark scenarios to real-simulation topology. These thorough and meticulous verifications establish the necessary evidence that the Twol-amalgamated priority queues are capable and useful for enhancing the performance of current priority queues.

The rest of this paper is organized as follows. Section 2 describes in detail the Twol algorithm, Section 3 provides the theoretical analysis of the complexity of the Twol-amalgamated priority queues, Section 4 discusses the measurement methods and benchmarking used in the experiments, Section 5 presents the empirical results, and Section 6 concludes this paper.

2. TWO-TIER LIST-BASED STRUCTURE (TWOL)

As shall be discussed, Twol has some similarities and differences compared to the CQ. Therefore, we shall start off by briefly discussing how a CQ priority queue structure works.

A CQ has M buckets numbered 0 to $M-1$, a bucketwidth δ, and an absolute start time t_s of bucket[0]. For each event e in the CQ, event timestamp $t(e) \geq t_s$, and the event e is located in bucket i, where $i = \lfloor (t(e) - t_s)/\delta \rfloor \bmod M$. We can also describe the CQ structure using an analogy of a circular-year desk calendar by the following: there are M days in a year each of duration δ which started at absolute time t_s and each event is placed on the day it is to occur regardless of the year. The duration of a year is $M\delta$ and therefore the time span of year j is $\big[t_s + jM\delta\ ,\ t_s + (j+1)M\delta\big)$. For practical implementation, a CQ has two important parameters, *bottom* and *top*; $bottom = M/2$ and $top = M \cdot 2$. They represent the lower and upper threshold, respectively, of the number of events which can exist in the CQ before a restructuring process takes place to modify M and δ. If *top* threshold is breached, that is, there are at least $M \cdot 2 + 1$ events, then $M_{\text{new}} = M \cdot 2$, $bottom = M$ and $top = M_{\text{new}} \cdot 2$. The bucketwidth δ is obtained via sampling some events in the CQ to estimate the average interevent time interval among all the events.

As an example, there are $N = 8$ events, $M = 4$, $\delta = 40$, and $t_s = 10$ (see Table I). The eight events have timestamps 10, 51, 70, 100, 127, 170, 226, and

Table I. Arrangement of Events in CQ Example

Bucket Index	0	1	2	3
Year[0] [10, 170)	10	51 70	100 127	—
Year[1] [170, 330)	170	226	265	

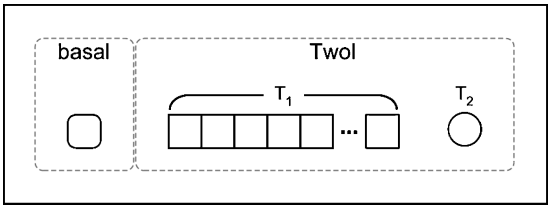

Fig. 1. Twol-amalgamated priority queue structure.

265. In this example, the next event to be deleted and processed has timestamp 10. Since 170 is greater or equal than maximum time span of year[0], it will not be processed until all the year[0] events are processed. Therefore, next to be examined is bucket[1]. Suppose processing of events in year[0] does not generate new events. After processing event 127, since bucket[3] is empty, the next to be examined is bucket[0] and event 170 is deleted. Right after event 226 is deleted, number of events in the CQ has breached the *bottom* threshold, that is, $N < M/2$, where $N = 1$. Thus a restructuring process takes place to modify $M_{\text{new}} = M/2 = 2$ and δ_{new} is some predefined number since there is only one event left. This completes our review of the CQ data structure.

We shall now delve into the Twol structure. Figure 1 shows the structure of a Twol-amalgamated priority queue. It comprises two fundamental structures, the basal and the Twol. The basal structure can be made up of any priority queue implementation. In this paper, we have implemented the Twol structure with three basal priority queues, namely, the Henriksen's queue, splay tree, and skew heap, for benchmark purpose. The rest of this section describes in detail the novel Twol structure.

The efficient design of the Twol structure is made up of two tiers. The first tier (T_1) is made up of an array of buckets where each bucket contains an unsorted linked list. The second tier (T_2) is simply an unsorted linked list. The structure of Twol is based on an improvisation of the CQ structure. The reason in choosing a list-based structure is due to the well-known fact that tree-based algorithms are bounded by O(log(n)), whereas a list-based structure such as the CQ has the potential to achieve near O(1) in good operating conditions.

Twol differs significantly from the CQ structure in three areas. Firstly, in the CQ (Brown 1988), the events in each bucket sublist are in sorted time order. The insertion of events can sometimes be an inefficient and enduring process. This can happen in situations where the event distribution is heavily skewed, leading to some very long bucket sublists which can lead to near O(n) insertion

cost. For the Twol structure, the linked lists in both T_1 and T_2 are unsorted. Each Insert(e, S) appends the event e to an unsorted linked list, leading to O(1) insertion cost.

Secondly, the CQ employs a resize operation that is performed according to the ratio of the number of events to the number of buckets. Though this operation operates efficiently for some priority increment distributions, it falters in scenarios where there are excessive resize function calls. This can occur when the ratio of events to buckets varies widely, for example, in battlefield simulation scenarios [Rönngren and Ayani 1997]. The Twol structure does not require a resize function.

Thirdly, the performance of the CQ is shown to have constant amortized behavior when the events are spread evenly. However, the CQ can degrade to O(n) complexity when a majority of the events fall in a single or in a few buckets. For this proposed Twol structure, it manages to offer near constant amortized behavior whilst keeping its bound at O(log(n)) amortized complexity upon amalgamation with tree-based basal structures, or O($\sqrt{n}$) for the Henriksen's queue.

Twol overcomes the shortcomings of the CQ in several aspects. Firstly, Twol postpones the sorting of events within the bucket sublists in T_1 until they are transferred to a basal structure where they are sorted. Each Insert(e, S) operation merely determines the correct bucket index to append event e to the bucket's sublist without sorting. This gives a majority of Insert operations an O(1) access time. Secondly, Twol does not utilize a resize operation. It obtains the parameters of T_1 from the dynamic variables in T_2, which grant an approximation of the event distribution in T_2.

Before we present a detailed description of the Twol algorithm in the following subsections, the purpose/definition of several important variables associated with the structure are stated as follows:

T_1_*Start*: used to calculate the bucket index of an event to be inserted in a particular bucket.

T_1_*Cur*: the minimum timestamp of the event which should be inserted in the Twol structure. If T_1_$Cur \leq t(e) < T_2$_Cur, where $t(e)$ is the timestamp of an event e, e is to be inserted in T_1.

T_1_*Bw*: width of a bucket in T_1.

T_2_*Min*: the minimum timestamp of events in T_2.

T_2_*Max*: the maximum timestamp of events in T_2.

T_2_*Cur*: the minimum timestamp of events which must be inserted in T_2.

T_2_*Num*: the number of events in T_2.

2.1 The Twol Algorithm—DeleteMin(S) Operation

At the onset, the basal structure and the Twol structure (both T_1 and T_2) are empty. All Insert(e, S) operations insert events into T_2, that is, Insert(e, T_2), without a linear search for the correct event position. Events are simply appended to the unsorted linked list T_2. On the first DeleteMin(S), all the events in T_2 are transferred to T_1. The bucketwidth of T_1 is determined using the

expression:

$$\text{Bucketwidth of } T_1 = T_1\_Bw = (T_2\_Max - T_2\_Min)/T_2\_Num, \tag{1}$$

and the bucket index in which an event is inserted in T_1 is determined similarly to CQ:

$$\text{Bucket index} = \lfloor (t(e) - T_1\_Start)/T_1\_Bw \rfloor. \tag{2}$$

The rationale of using Eq. (1) to obtain the bucketwidth is that Twol makes an assumption that the interevent timestamp is uniformly distributed. Hence, each bucket should contain one event on average if the distribution is uniform. If the assumption of uniform distribution is not true, there would exist some buckets that contain many events. However, the performance is not very much affected even if some buckets contain many events. This is because subsequent insertion cost of events into T_1, or Insert(e, T_1), is still O(1) since events are simply appended to the sublists without sorting. After obtaining T_1_Bw, both T_1_Start and T_1_Cur are set to T_2_Min, and T_2_Start is set to T_2_Max. The bucket index of all the events inserted into T_1 during a transfer from T_2 to T_1 adheres to Eq. (2), which is similar to an event inserted in a CQ but without requiring modulus by the number of buckets, unlike in a CQ. This is different from a CQ, because in a CQ there is no dedicated overflow structure but rather the overflow events are inserted into the sublists, forming multiple *years*, where a year is the time interval that spans the total length of all the bucket sublists. For T_1, there exists only one year.

Thereafter, the buckets in T_1 are processed on a bucket-by-bucket basis, starting with the first nonempty bucket of T_1. If there is only one event in a bucket, the event is simply deleted since it is the event with the minimum timestamp, thereby bypassing the basal structure. If, however, there are more than one event in a bucket, the events are to be inserted into the basal structure using the basal priority queue's native Insert operation. And thereafter, the basal priority queue's native DeleteMin operation is carried out. After the events in the basal structure are depleted, the second nonempty bucket in T_1 is processed and so on, until both the basal structure and T_1 are empty. Under that circumstance, a transfer of events from T_2 to T_1 is performed again. We shall illustrate the DeleteMin(S) of the Twol algorithm in the following example.

Let n_B be the number of events in the basal structure, n_{T1} and n_{T2} be the number of events in T_1 and T_2 of Twol, respectively, n_{Twol} be the number of events in the Twol structure $= n_{T1} + n_{T2}$, and the total number of events $n = n_B + n_{\text{Twol}}$. As an example, assume $n = 0$ initially. Insert eight events that have timestamps 36, 10, 51, 100, 25, 175, 226, 127 according to this order. These eight events will be appended to the head of T_2 without sorting, and the variables of T_2 will be updated as $T_2\_Min = 10$, $T_2\_Max = 226$, $T_2\_Num = 8$, $S = \{127, 226, 175, 25, 100, 51, 10, 36\}$, where event with timestamp 127 is now at the head of T_2. On the first DeleteMin(S) operation, because n_B and $n_{T1} = 0$, a transfer of events from T_2 to T_1 is necessary. T_1's variables are initialized: T_1_Start and $T_1\_Cur = 10$, and from Eq. (1), $T_1\_Bw = (226 - 10)/8 = 27$. The events are then inserted in the corresponding bucket using Eq. (2). For instance, the event with timestamp 36 is inserted into the bucket index $= \lfloor (36 - 10)/27 \rfloor = 0$.

Table II. Arrangement of Events in T_1 of Twol

Bucket Index	0	1	2	3	4	5	6	7	8
	36 10 25	51	—	100	127	—	175	—	226

After the transfer, the events in T_1 will be arranged as seen in Table II. Also, T_2_Cur is then tuned to be T_2_Max (i.e., 226) so that additional events to be inserted and which timestamps are greater or equal to T_2_Cur (i.e., 226) will fall in T_2 and not in T_1 nor the basal structure. Finally, the events {36, 10, 25} are then transferred from the first nonempty bucket, Bucket[0], to the basal structure in a stable FIFO manner. T_1_Cur is then set to $T_1\_Start + T_1\_Bw = 37$. Thereafter, the event timestamp 10 is then subsequently deleted using the basal structure's DeleteMin(S_B), where S_B is the set of events in the basal structure = {10, 25, 36}.

After events with timestamps 25 and 36 are deleted and assuming there is no additional Insert operation, the subsequent DeleteMin operation will then process the next nonempty bucket, Bucket[1], which contains one event with timestamp 51. Since there is only one event, the DeleteMin operation will simply delete this event to be processed bypassing the basal structure for more efficiency. That is, the greater the number of buckets that contain only one event, the more efficient is the Twol algorithm.

2.2 The Twol Algorithm—Insert(e, S) Operation

For each Insert(e) operation, the timestamp associated with event e, $t(e)$, is compared with the threshold values of T_1 and T_2 to find out where it should be inserted. If $t(e) \geq T_2\_Cur$, e is inserted in T_2 and the variables T_2_Min, T_2_Max, and T_2_Num are updated accordingly. Otherwise if, $T_1\_Cur \leq t_e < T_2\_Cur$, it is inserted in T_1 where the bucket index is given as shown in Eq. (2). Finally, if e is not placed in T_1, it is then inserted into the basal structure.

From the example given in Section 2.1, after the DeleteMin(S_B) operation where the event timestamp 10 is removed, suppose as an example three events with timestamps 250, 105, 30 are inserted. Since timestamp 250 is greater than T_2_Cur (i.e., 226), the corresponding event is appended to T_2. Using Eq. (2), the event with timestamp 105 is inserted in T_1 into the sublist with bucket index = $\lfloor(105 - 10) / 27\rfloor = 3$. Finally since timestamp 30 is less than T_1_Cur (i.e., 37), it is inserted in the sorted basal structure.

3. THEORETICAL ANALYSIS OF TWOL-AMALGAMATED PRIORITY QUEUES

In this analysis, in addition to the usual analysis of the upper bound using the average-case and worst-case complexity, the amortized complexity [Tarjan 1985] is also employed. The amortized complexity of an operation is to compute the sum of the costs of a *sequence* of n operations and divide this sum by n operations, where the cost is the time taken. With a combination of these

interpretations, we define *expected amortized complexity* of a priority queue as the combination of traditional amortized complexity with an expectation analysis on the mean number of events in the priority queue that it will maintain in nondecreasing sorted time-order at any time for N total number of events in the priority queue. The *worst-case amortized complexity* is defined as the combination of traditional amortized complexity, and the worst-case maximum number of events that a priority queue has to maintain, as the total number of events in the priority queue increases. To prove the expected and worst-case amortized complexity of a priority queue, we only have to consider the operations involved in the priority queue, which are mainly the Insert and DeleteMin operations.

LEMMA 1. *A Twol-amalgamated priority queue has* O(1) *expected amortized complexity if the average number of events in the basal structure is bounded by a constant.*

PROOF. This can be shown true by considering the cost of an Insert operation and the cost of a DeleteMin operation in a Twol-amalgamated priority queue (made up of a basal structure and a Twol structure). If the costs of these two primary operations in the Twol-amalgamated priority queue are O(1), then the queue is also O(1). There are two types of costs to be considered: fixed cost and variable cost.

Insert cost: There are two scenarios. Firstly, if an event is inserted in the Twol structure, the fixed cost incurred is O(1). The reason is as follows. The Twol structure is made up of two parts, T_1 and T_2. T_2 of the Twol structure is an unsorted linked list. Each insertion of an event in T_2 simply appends the event to T_2, resulting in a fixed O(1) cost. T_1 is made up an array of buckets where each bucket holds a sublist. A sublist is made up of an unsorted linked list. Each insertion of an event in a sublist in T_1 incurs a fixed O(1) cost of calculating the correct bucket index (see Eq. (2)) and thereafter, appending the event to the sublist. Secondly, if an event is to be inserted in the basal structure, the Insert(e, basal) operation can incur a variable cost. Therefore, if the average number of events in the basal structure is bounded by a constant irrespective of N, this Insert(e, basal) variable cost will also be bounded by a constant.

DeleteMin cost: There are three scenarios. Firstly in Scenario_1, if the basal structure has events, the events are removed via the basal structure's native DeleteMin(S_B) operation, where S_B is the set of events in the basal structure. This incurs a variable cost that is dependent on the number of events n_B in the basal structure, as well as the complexity of the DeleteMin(S_B) operation (e.g., O(log(n_B)) for a skew heap and O(1) for a splay tree). Secondly in Scenario_2, if the basal structure is empty and the first-tier T_1 has events, the events from T_1 are transferred to the basal structure on a bucket-by-bucket basis. This incurs a variable cost depending on the number of events n_B to be placed in the basal structure, as well as the complexity of the basal structure's native Insert(e, basal) operation. Thereafter, Scenario_1 proceeds. Lastly in Scenario_3,

if the basal structure and the first-tier T_1 of the Twol structure are empty, a fixed cost is incurred to transfer events from T_2 to T_1. Suppose that N events in T_2 are transferred to T_1; a fixed O(1) cost per event is incurred since each event merely calculates the bucket index and then is appended in the corresponding bucket's sublist. Since the insertion of an event in a sublist only involves appending and not performing a sequential search for the correct position, the total one time cost of this entire T_2 to T_1 transfer is O(N). And amortizing this O(N) total cost over N events, we get O(1). Thereafter, $\mathsf{Scenario}_2$ proceeds. Note also that there is a fixed O(N) cost of traversing from the first to the last bucket in T_1 for N events transferred from T_2 to T_1 and subsequently from T_1 to the basal structure. Amortizing this cost of skipping O(N) buckets over N events, we get O(1) as well. As shown later in Lemma 5, the upper bound of the number of buckets in T_1 is $N + 1$. Therefore in these three scenarios, if the expectation of n_B is bounded by a constant, then regardless of the complexity of the Insert(e, basal) and DeleteMin(S_B), the variable costs will be O(1) amortized. □

LEMMA 2. *The average number of events in the basal structure is bounded by a constant if the average number of events in the first bucket in the first-tier T_1 of the Twol structure is bounded by a constant.*

PROOF. Assuming that at the onset, both the basal structure and T_1 are empty whereas T_2 contains N events. Upon a DeleteMin operation, T_2 undergoes a transfer process in which all the N events are transferred to $N + 1$ buckets in T_1. Subsequently, the events in T_1 are transferred to the basal structure on a bucket-by-bucket basis. During the first DeleteMin operation, the first bucket (Bucket[0]) will transfer all the events in its sublist to the basal and the minimum timestamp event is removed from the basal structure. To simplify subsequent analysis, the basal structure can be conceptualized as always corresponding to the first bucket in T_1 and which contains the basal structure. Subsequently after the basal structure is empty and assuming Bucket[1] is nonempty, Bucket[1] will transfer its events to the basal structure. Bucket[0] can now be deemed as deleted, and Bucket[1] can now be considered as the first bucket in T_1. Therefore, if the expected number of events in the first bucket in T_1 is bounded by a constant, the average number of events in the corresponding basal structure is certainly also bounded by a constant. □

The rest of this section proceeds to discuss the proof that under reasonable assumptions, the average number of events in any bucket in T_1 is indeed bounded by a constant. The proof is very much simplified due to the Twol structure's uncanny resemblance to an implementation of the CQ which employs the unsorted linked list bucket discipline. We call this implementation the UCQ, so as to differentiate it with the standard CQ by Brown [1988]. We shall discuss the similarity between UCQ and the Twol structure. Thereafter, we present a theoretical analysis of the UCQ given in Erickson et al. [2000] and extend it to the Twol algorithm.

3.1 Similarities and Differences of the Twol Structure to the UCQ

List-based queues can be implemented in which each linked list in the bucket (sublist) is unsorted as discussed by Erickson et al. [2000]. We call this implementation the UCQ.[1] Erickson et al. have employed Markov chain analysis to investigate the performance of UCQ under a *static* scenario where the number of active events N in the UCQ remains a constant. Generally, the number of active events may vary over time. However, the static case is an important case because it is commonly encountered in discrete event simulations. For instance, when simulating a parallel computer, if there are N processors, then there are exactly N active events in the priority queue [Erickson et al. 2000]. Another example is when we use the process-oriented Swan simulator [Thng and Goh 2004] to simulate computer network topologies; if there are N nodes, then there are exactly N active events in the priority queue.

Erickson et al. [2000] have proven theoretically that under a static scenario where the number of events N in the UCQ remains a constant, the UCQ has constant expected processing time per event if the mean of the *jump* variable is a constant. The jump, which has the value $t(e) - t(e_0)$, is a random variable sampled from some priority increment distribution that has a mean μ, where e is a newly generated event with time $t(e)$ and e_0 is the current minimal time. They proved that this expected processing time depends on the μ and not on the shape of its probability density. They derived the expected time per event for the infinite bucket UCQ (UCQ_i) and the finite bucket UCQ (UCQ_f).

The best theoretical performance occurs in the UCQ_i where there are an infinite number of buckets, where the array of buckets in the UCQ_i spans one year time interval. The performance of the UCQ_f is at best equal to that of the UCQ_i. The events in the UCQ_i are processed bucket by bucket, unlike in the UCQ_f where there are a finite number of buckets. In a UCQ_f, the buckets can contain events which are in the future years and thus will be processed only much later. This results in a poorer performing UCQ since the DeleteMin operation has to search through these future years events. It is also likely that after searching an entire bucket of events, there is no event with the minimum timestamp because all the events in the bucket are in the future years.

We notice that their optimized UCQ suffers from two limitations. Firstly, the unsorted bucket discipline has an $\Omega(n_x)$ cost per DeleteMin operation, where n_x is the number of the events in the bucket from which the event is deleted. Secondly, in simulation models where N and/or μ can vary with time, the UCQ must undergo costly resize operations to obtain a new bucketwidth δ and a new number of buckets used. Otherwise, its performance will degrade to $\text{O}(N)$. Nevertheless, it is a well-written theoretical article which provided insights into improving the performance of the UCQ.

[1]This is different from Brown's CQ [1988] which employs a sorted list in each bucket. The UCQ has constant insertion cost but a guaranteed $W(n_x)$ deletion cost, where n_x is the number of events in a sublist. The CQ has worst-case $\text{O}(n_x)$ insertion cost while it has constant deletion cost. Hence on average, the UCQ has higher amortized cost compared to the CQ.

Coincidentally, the UCQ bears a strong resemblance to the first-tier T_1 of the Twol structure. The UCQ and T_1 are both made up of an array of buckets where each bucket contains an unsorted linked list. Therefore, inserting an event in either the UCQ or T_1 is just appending the event to the sublist, leading to an O(1) cost per insertion. Furthermore the UCQ and T_1 of Twol structure similarly distribute the events among the buckets according to the same rule, that is, Eq. (2). There is however a distinct difference between the UCQ and T_1, and this is discussed in Lemma 3.

LEMMA 3. *The number of events n_{T1} in the first-tier T_1 of the Twol structure is always less than the number of events N in the UCQ and $n_{T1} \rightarrow 0$ as time progresses.*

PROOF. The structure of the UCQ and T_1 is similar in nature. However, the UCQ is guaranteed to contain all the N events whereas T_1 does not. The Twol structure is made up of two tiers, T_1 and T_2. After every transfer of events from T_2 to T_1, T_1 contains N events. At this point, the number of events in T_1 is equal to the UCQ. Immediately after this, n_B events from the first nonempty bucket of T_1 are transferred to the basal structure. Since $N = n_B + n_{T1} + n_{T2}$, the number of events n_{T1} in T_1 is always less than N. The events in the basal structure are then deleted, and new events are inserted in either the basal structure or the Twol structure. Subsequently, the rest of the buckets in T_1 are processed on a bucket-by-bucket basis. As time passes, the rest of the buckets in T_1 will eventually be transferred to the basal structure until T_1 becomes empty. However for the static UCQ, the number of buckets that contain events remains relatively constant and the expected number of events in each bucket remains the same. □

Lemma 3 provides the justification that the theoretical analysis of the Twol structure is for the worst-case scenario to that considered for the UCQ. We assume for the worst case that all newly events are inserted in T_1 and none in T_2 or the basal structure even as time progresses, that is, $n_{T1} = N$. With this, we can proceed to use Lemma 4 to derive Theorem 1.

3.2 Applicable Lemmas and Theorem for the Twol

Due to the similarity of the UCQ and the first-tier T_1 of the Twol structure, we now proceed to use the previous result derived for the UCQ under a hold scenario where the number of events in the queue do not vary. We define the following variables:

- B: random variable that the first bucket contains a certain number of events ranging from 0 to N.
- δ: bucketwidth of T_1.
- μ: finite mean of the jump random variable that defines the priority increment distribution of the N events. The jump random variable has density function f and cumulative density function denoted by $F(x)$, where $F(x) = \int_0^x f(z)dz < 1$.

q_i: limiting probability that a bucket has exactly i events inserted.
E_N: limiting expected number of events in a bucket is defined as

$$E_N(\delta) = \sum_{j=1}^{N} j q_j, \tag{3}$$

since the limiting probabilities q_i for $0 \le i \le N$ exist and are independent of the initial state of queue [Erickson et al. 2000].

LEMMA 4 (ERICKSON ET AL. [2000]). *For $N \ge 2$ and all $\delta > 0$, the N events are distributed amongst the T_1 of the Twol structure, UCQ-like, according to*

$$q_0 = \frac{\mu}{\mu + N\delta}, \tag{4}$$

and for $j = 1, 2, \ldots, N$, we have

$$q_0 B(j) \le q_j \le \frac{q_0 B(j)}{1 - F(\delta)}, \tag{5}$$

where $B(j)$ is the tail of the binomial distribution for N trials with "success" parameter p:

$$B(j) = \sum_{k=j}^{N} \binom{N}{k} p_k (1-p)^{N-k}, \tag{6}$$

$$p = p(\delta) = \frac{1}{\mu} \int_0^{\delta} [1 - F(x)] dx, \tag{7}$$

where $\delta[1 - F(\delta)]/\mu \le p \le \delta/\mu$.

Remark. Lemma 4 provides the mathematical foundation for the UCQ under a hold scenario. The relevant proof is presented by Erickson et al. [2000] using a Markov chain model.

THEOREM 1. *The expected number of events $E_N(\delta)$ in any bucket in T_1 of the Twol structure is bounded above by*

$$\frac{4\mu c + c^2}{4\mu(\mu + c)[1 - F(c/2)]}, \tag{8}$$

where c is a constant.

PROOF. Since

$$\sum_{j=1}^{N} j B(j) = \frac{1}{2}[(N^2 - N)p^2 + 2Np],$$

and from (3) and (5), we have

$$\begin{aligned} E_N(\delta) &= \sum_j j q_j \le \frac{q_0}{1 - F(\delta)} \sum_j j B(j) \\ &= \frac{\mu}{(\mu + N\delta)[1 - F(\delta)]} \sum_j j B(j) = \frac{\mu}{2(\mu + N\delta)[1 - F(\delta)]} [(N^2 - N)p^2 + 2Np]. \end{aligned}$$

Since $\delta[1-F(\delta)]/\mu \leq p \leq \delta/\mu$, and noting that $\delta = c/N$, where $c = T_2\_Max - T_2\_Min$ [see (1)] is some constant independent of N,[2] we have

$$\begin{aligned} E_N(\delta) &= \sum_j jq_j \leq \frac{\mu}{2(\mu+N\delta)[1-F(\delta)]}[(N^2-N)p^2+2Np] \\ &\leq \frac{\mu}{2(\mu+N\delta)[1-F(\delta)]}\left[(N^2-N)\left(\frac{\delta}{\mu}\right)^2+2N\frac{\delta}{\mu}\right] \\ &= \frac{\mu}{2(\mu+c)[1-F(c/N)]}\left[(N^2-N)\left(\frac{c}{N\mu}\right)^2+\frac{2c}{\mu}\right]. \end{aligned}$$

Since $N \geq 2$ in order for T_1 to exist (if $N = 1$ then a DeleteMin proceeds immediately from T_2 without the need to construct T_1),

$$\begin{aligned} E_N(\delta) &\leq \frac{1}{2(\mu+c)[1-F(c/2)]}\left[\frac{c^2}{\mu}-\frac{c^2}{2\mu}+2c\right] \\ &= \frac{4\mu c+c^2}{4\mu(\mu+c)[1-F(c/2)]}, \end{aligned}$$

where $F(c/2) < 1$. □

This proves that under a hold scenario, if μ does not vary, $E_N(\delta)$ is bounded by an explicit constant where T_1 contains all the N events, that is, $n_{T1} = N$ and $n_B + n_{T2} = 0$. However, Lemma 3 proves that this is the worst-case scenario since n_{T1} is always less than the number of events N in the UCQ. Moreover, as time progresses, n_{T1} decreases to zero. Thus the bound given in Eq. (8) will also decrease gradually to zero, before a transfer of events from T_2 to a new T_1 is initiated.

COROLLARY 1. *A Twol-amalgamated priority queue has O(1) expected amortized complexity under a hold scenario where the mean of the jump μ remains a constant.*

PROOF. The proof is obtained by combining Lemma 1 and Theorem 1. □

LEMMA 5. *The number of buckets used in T_1 is bounded above by $N+1$ for all scenarios.*

PROOF. From Eqs. (1) and (2), it is obvious that the number of buckets is equal to $T_2\_Num + 1$, where the last bucket contains the event with the maximum timestamp. Considering for the worst-case scenario where all the N events are inserted into T_2 and there are none in the basal structure or T_1, a DeleteMin operation will have to transfer N events from T_2 to T_1, where $T_2\_Num = N$. □

Lemma 5 illustrates the fact that a Twol-amalgamated priority queue can achieve Corollary 1 using only an explicit upper bound of $N+1$ buckets in T_1.

[2] T_2_Max-T_2_Min represents the span of the priority increment distribution which does not vary even if N varies.

Table III. Performance of Priority Queues

Queue	Insert Amortized (Expected[a], Worst-case)	Insert max	DeleteMin Amortized (Expected[a], Worst-case)	DeleteMin max
Skew Heap	O(log(n))	O(n)	O(log(n))	O(n)
Splay Tree	O(log(n))	O(n)	O(1)	O(1)
Henriksen's Queue	O(log(n)), O($\sqrt{n}$)	O(n)	O(1)	O(1)
CQ, DCQ	O(1)[a], O(n)	O(n)	O(1), O(n)	O(n)
Twol-Amalgamated Skew Heap	O(1), O(log(n))	O(n)	O(1), O(log(n))	O(n)
Twol-Amalgamated Splay Tree	O(1), O(log(n))	O(n)	O(1)	O(n)
Twol-Amalgamated Henriksen's	O(1), O($\sqrt{n}$)	O(n)	O(1)	O(n)

[a]Only Twol-amalgamated priority queues are theoretically proven, with reasonable assumptions, to have this expected amortized complexity. For the rest of the queues, this expected value is observed from many empirical studies such as Rönngren and Ayani [1997] and by the authors as shown in Section 5.

COROLLARY 2. *The worst-case amortized performance of a tree-based Twol-amalgamated priority queue is bounded above by* $O(\log(N))$.

PROOF. From Corollary 1, we know that a Twol-amalgamated priority queue has O(1) expected amortized complexity if μ do not vary. However in situations where μ does vary, the Twol-amalgamated priority queue is expected to perform reasonably better than a single basal structure if the probability of insertion of events in T_1 and T_2 is greater than zero (i.e., probability of insertion into the basal structure is less than 1). In scenarios where the μ varies, such as in a camel or change distribution (see Section 4.2), if μ is large so that subsequent events occur in T_1 or T_2 and not in the basal structure, the performance is not much affected since the cost of insertion in T_1 or T_2 is O(1). Moreover, as the buckets in T_1 are transferred to the basal structure until all the buckets in T_1 are empty, a new set of T_1 parameters are dynamically generated according to the variables in T_2. The worst-case scenario in a Twol-amalgamated priority queue occurs when all the N events fall in the basal structure (this is highly unlikely if the jumps are greater than zero, i.e., events do not have equal timestamp). The performance of a tree-based Twol-amalgamated priority queue, where the basal structure is a skew heap or a splay tree, will then decline from O(1) to O(log(N)), which is the native performance of a traditional skew heap or splay tree. □

Table III summarizes the theoretical performance of the priority queues.

4. PERFORMANCE MEASUREMENT TECHNIQUES

The performance of priority queues is often measured by the average access time of Insert(e, S) or DeleteMin(S) under different load conditions. The parameters to be varied for each priority queue performance benchmark are the access pattern, the priority increment distribution, and the queue size. The queue size is defined as the total number of events in a priority queue. For the case of Twol-amalgamated priority queues, the queue size refers to the sum of the

events in both the basal and Twol structures. We shall discuss the other two parameters in the following subsections.

4.1 Access Pattern Models

The access pattern models that have been proposed either emulate the steady-state or the transient phase of a typical simulation. They are as follows:

Classic Hold Model [Jones 1986]. The queue to be tested is initially built up to the specified benchmark size by using a random series of Insert(e, S) and DeleteMin(S), where the Insert operation has a slightly higher probability than the DeleteMin. Thereafter, a series of hold operations ensue. A hold operation is defined as a DeleteMin followed by an Insert. The timestamp of a newly inserted event is obtained as follows: A random variable that describes the desired priority distribution to be tested is picked. The value of the random variable is then added to the timestamp of the event that was just deleted, and the result is the timestamp of the newly inserted event. The average access time to be calculated is the average time taken for one hold operation. The recommended number of hold operations, in order to obtain a reasonable accurate average, should be 30 times the queue size [Rönngren and Ayani 1997]. This method has the following advantages:

(1) The problem of determining the transient period is avoided.
(2) The effects of the transient period on the different queue sizes tested are similar [Rönngren and Ayani 1997].

Up/Down Model [Rönngren et al. 1993]. In an up/down test, the queue is built up to the benchmark queue size by a sequence of Insert operations. Thereafter, the queue size is returned to zero by an equally long sequence of DeleteMin operations. The average access time to be calculated is the time taken for all queue operations (in the insertion phase and deletion phase), divided by the total number of queue operations. This up/down model emulates the worst-case scenario of a simulation job which is not stable in queue size [Rönngren et al. 1993].

4.2 Priority Increment Distributions

The priority increment distributions, often used for the benchmarking of priority queue structures, are found in Table III, where rand() returns a random number [Park and Miller 1988] in the interval [0,1]. The camel(x,y) distribution [Rönngren et al. 1993] represents a 2-hump heavily skewed distribution with x% of its mass concentrated in the two humps and the duration of the two humps is y% of the total interval. In addition to the eight distributions as described in Table III, the Change(A, B, x) distribution [Rönngren et al. 1993] was also used to test the sensitivity of the CQ when exposed to drastic changes in priority increment distribution. The compound distribution change(A, B, x) interleaves two different priority increment distributions A and B together. Initially, x priority increments are drawn from A followed by another x priority increments drawn from B and so on. Change

Table IV. Priority Increment Distributions

Distribution	Expression to Compute Random Number	Bias
Exponential(1)	**if** `(x = rand()) == 0` **then** `infinity` **else** − `ln(rand())`	0.50
Uniform(0, 2)	`2*rand()`	0.66
Uniform(0.9, 1.1)	`0.9 + 0.2*rand()`	0.96
Bimodal	9.95238*rand() + **if** `rand() < 0.1` **then** `9.5238` **else** `0`	0.34
Triangular(0, 1.5)	`1.5*sqrt(rand())`	0.80
NegativeTriangular(0, 1000)	`1000*(1-sqrt (1-rand()))`	0.60
ExponentialMix	**if** `(rand() < 0.9)` **then** `Exponential(1)` **else** `Exponential(100)`	0.10
Camel(0, 1000, 0.001, 0.999)	See [Rönngren et al. 1993]	0.74

Table V. Benchmarking Platforms

Specifications	Intel Pentium 4 2.4B GHz	Intel Itanium 2 1.3 GHz—SGI Altix 3300	SGI MIPS R16000 700 MHz—SGI Onyx4	AMD Athlon MP 1.2 GHz
No. of Processors	1	8	4	2
L1/L2/L3 Cache	20 KB/512 KB/0 KB	32 KB/256 KB/3 MB	64 KB/4 MB/0 KB	128 KB/256 KB/0 KB
Shared RAM	1 GB	8 GB	4 GB	1 GB
Operating System	Linux Mandrake 9.1 Kernel 2.4.21	Linux Redhat 7.2 Kernel 2.4.20	Irix 6.5.21	Windows XP Professional with Service Pack 1
C Compiler	GCC 3.2.2	Intel ECC Compiler 7.1	MIPSpro Compilers 7.30	Borland C++ 5.5
Timer	gettimeofday	gettimeofday	gettimeofday	QueryPerformance counter
Holds/UpDowns	10/10	30/10	30/10	30/30
Queue Size	10–1 million	100–10 million	100–1 million	100–1 million

distributions can be used to model simulations where the priority increment distributions vary significantly over different time periods, for example, battlefield simulations. The change distributions we used are change(exp(1), triangular(90000, 100000), 2000), and change(triangular(90000, 100000), exp(1), 10000). All the parameters associated with the distributions are identical to those used in Rönngren and Ayani [1997]. A bias [Jones 1986] value of 1 corresponds to a FIFO behavior and a value of 0 corresponds to a LIFO queue.

4.3 Benchmark Architecture and the Effect of Cache Memory

The performance of different priority queues was obtained by conducting experiments on four different hardware platforms running different operating systems as illustrated in Table V. The algorithms were sequentially executed even though the AMD and SGI servers support multiprocessors. It should be noted for the experiment conducted on the Intel Pentium 4, processor's L1 and L2 cache were turned off via the mainboard BIOS. The main reason for disabling cache memory is so that the experiments will be more definitive of whether the access time characteristics can be demonstrated to be truly O(1). The problem with experiments conducted with cache is that cache memory runs several orders faster than normal memory while its size is also very limited. Hence at small queue sizes where all the states of the simulation can fit into the cache,

the access time will be the fastest. However, for large queue sizes, the effect of caching is no longer significant and the access time will increase due to the increasing reliance on the slower shared RAM for handling the states of the experiment. Experiments conducted with cache will have what is known as "knee" effects on access time characteristics. The "knee" effect is defined as a sudden rise in access time characteristics. The first knee effect occurs at a small queue size value which causes the L1 cache to be utilized fully so that reliance on the slower L2 cache starts to increase. The second knee effect occurs when a larger queue size value is reached which fully utilizes L2 cache so that reliance on relatively slower main memory starts to increase. Amongst the four hardware platforms considered, only the BIOS option of the Pentium 4 hardware allows the disabling of cache. By disabling the cache, it is possible to verify empirically by simulation whether the Twol-amalgamated priority queues are indeed expected O(1) and bounded by the basal structure's amortized complexity. These experiments are however done at the price of extremely slow simulations.

Simulation studies involving slow hardware have been presented before where the number of operations that are performed under Classic Hold is far less than the recommended 30 times the queue size criteria. For example, in Rönngren and Ayani [1997], the number of operations was set to five times the queue size in view of an Intel 80386 based Sequent symmetry platform. It is noted that performance results obtained with Classic Hold operations numbering less than 30 times the queue size will be influenced by some transient effects for those priority increment distributions with bias less than 0.5, for example, the bimodal and ExponentialMix distributions. However, for priority increment distributions with bias greater than 0.5, the performance result will converge after the number of Classic Hold operations have reached 5 times the queue size Rönngren and Ayani [1997]. For our simulations conducted under the Pentium 4 platform without cache, we set the number of Classic Hold operations to 10 times the queue size for all the priority queue structures considered. The number of Classic Hold operations to conduct is determined by the slowest performing priority queue structure which happens to be the CQ. For the case of the Twol-amalgamated priority queues, it should be noted that whether the number of Classic Hold operations was set to 10 or 30 times the queue size, we were able to obtain results within a reasonable time span even without caching. Since the results look similar, we only present those results that were obtained using 10 times the queue size so that it is consistent with all other simulation results obtained for the other priority queues. For the Intel Itanium 2, SGI MIPS, and AMD platforms, the number of operations was set to 30 times the queue size for all the priority queues under study. On the Intel Itanium 2 platform (SGI Altix 3300) that has a vast RAM of 8 GB, we obtained results for queue sizes ranging from 100 to 10 million events. This truly extends the validity of the performance of the priority queues for very large-scale discrete event simulation on the contemporary 64-bit processor technology.

The codes used for the Henriksen's queue and splay tree were based on the Pascal code used by Jones [1986]. Skew heap implementation was based on the nonrecursive code given in Jones [1989]. CQ and DCQ were based on the

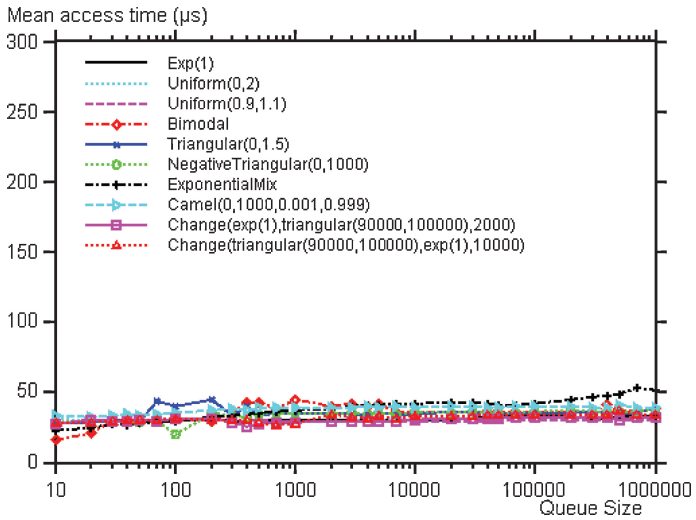

Fig. 2. Mean access time for Twol-amalgamated Henriksen's under Classic Hold experiments.

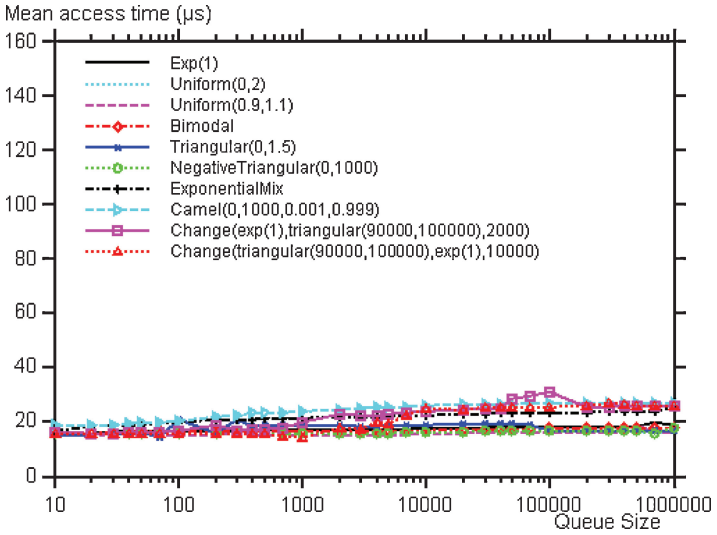

Fig. 3. Mean access time for Twol-amalgamated Henriksen's under up/down experiments.

codes supplied by Brown [1988], and Oh and Ahn [1998], respectively. Two empirical tests were conducted to verify that no items in the priority queues were gained or lost and that successive DeleteMin operations removed events in stable time-order.

The experiments were performed with the required memory for each priority queue being preallocated. This was to eliminate the underlying memory management system which might affect the results. This is a good practice in discrete event simulators as it prevents memory fragmentation when creating new

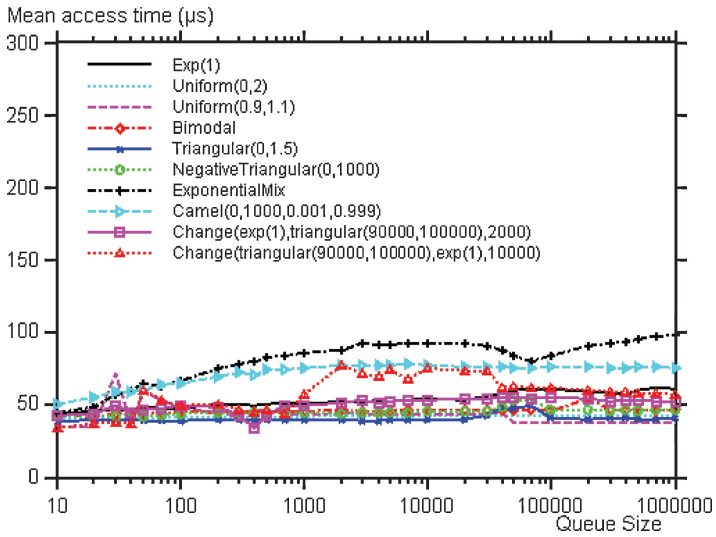

Fig. 4. Mean access time for Twol-amalgamated splay tree under Classic Hold experiments.

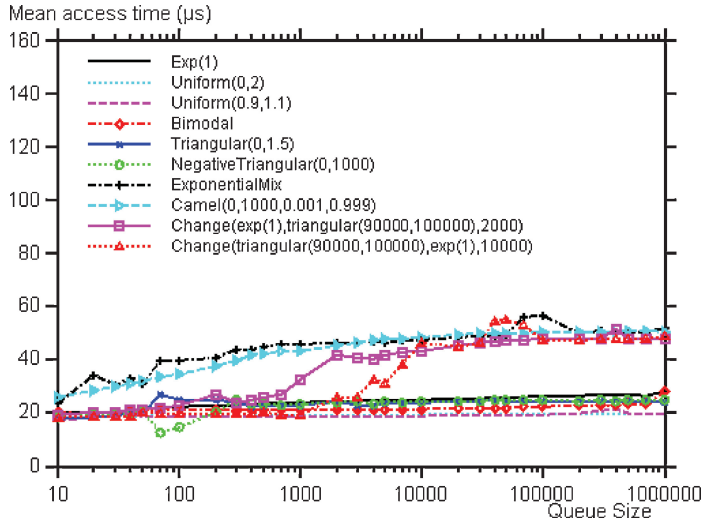

Fig. 5. Mean access time for Twol-amalgamated splay tree under up/down experiments.

events and deleting the serviced events. This method of preallocating memory would also enhance the performance of the simulators. The method of preallocation could be made dynamic by an initial preallocation and subsequently, an allocation of memory on demand methodology could be employed. All code was written in the C programming language with all recursive procedure calls and the like being eliminated. Loop overhead time and the time taken for random numbers generated were removed by factoring out the time required for running a dummy loop.

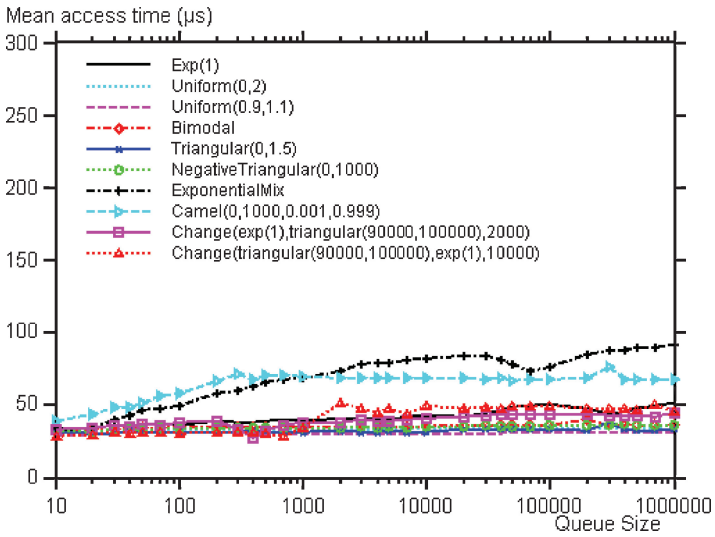

Fig. 6. Mean access time for Twol-amalgamated skew heap under Classic Hold experiments.

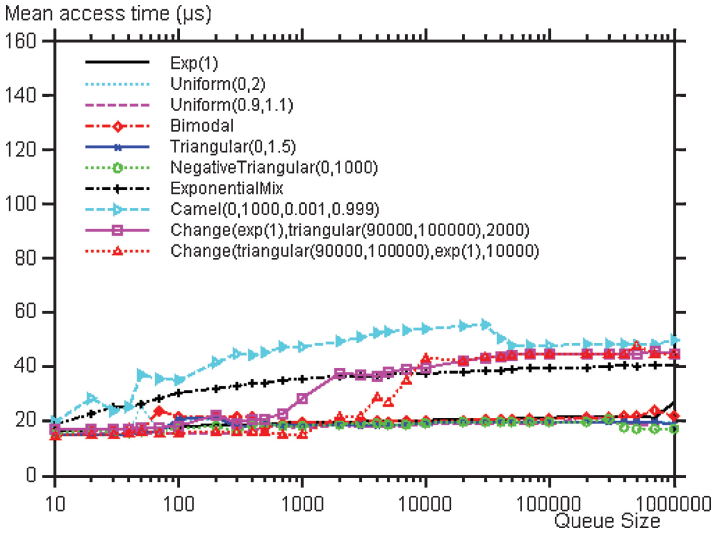

Fig. 7. Mean access time for Twol-amalgamated skew heap under up/down experiments.

5. EMPIRICAL RESULTS

In this section, we present empirical results concerning the performance of the various priority queue implementations. The results shown are the average values from five runs. A logarithmic scale is used for the queue-size axis leading to linear plots for logarithmic complexity. Results provided in Figures 2–17 are obtained from experiments conducted on the Intel Pentium 4 platform without caching whereas Figures 20–25 illustrate experimental results obtained from the Intel Itanium 2, SGI MIPS R16000, and AMD Athlon MP.

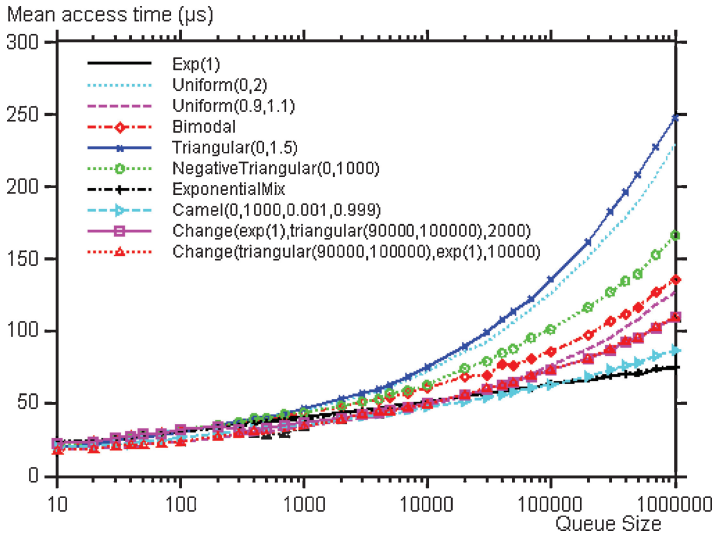

Fig. 8. Mean access time for Henriksen's queue under Classic Hold experiments.

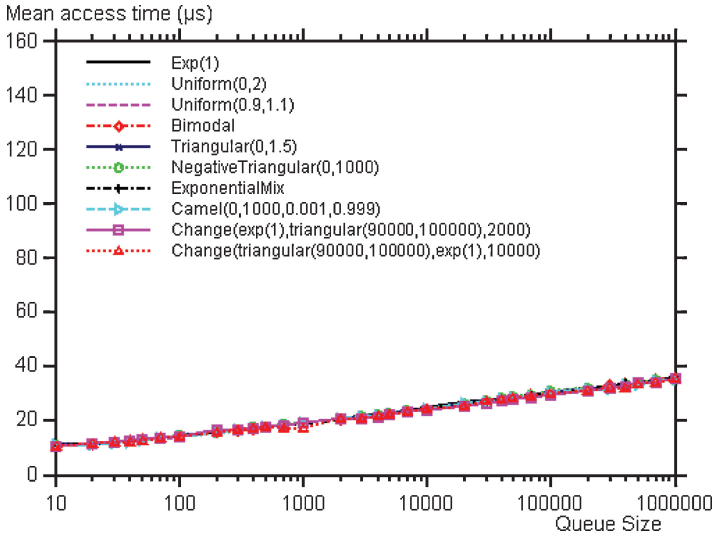

Fig. 9. Mean access time for Henriksen's queue under up/down experiments.

5.1 Performance on the Intel Pentium 4 (Cache Disabled)

Figures 2–7 reveal the performance of the priority queues after the amalgamation with the Twol structure under the Classic Hold and up/down experiments for queue size ranging from 10 to 1 million events and for the 10 different priority increment distributions. The performance of the Twol-amalgamated priority queues is as follows:

Amalgamated Henriksen's Queue: Figures 2 and 3 demonstrate that the amalgamated Henriksen performs very stably with near constant performance

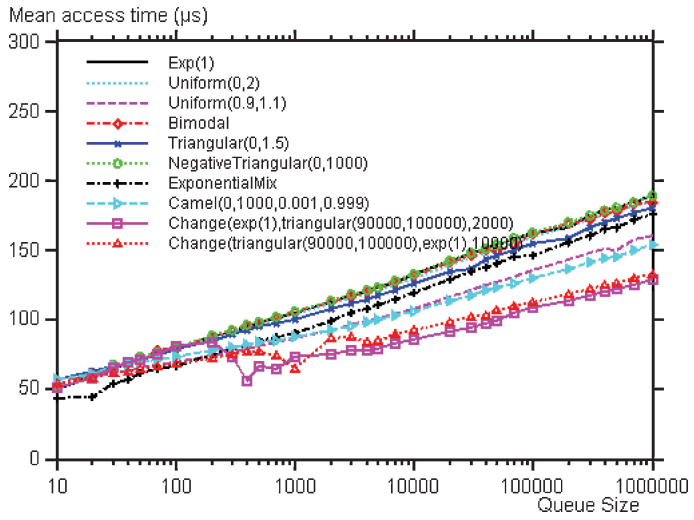

Fig. 10. Mean access time for splay tree under Classic Hold experiments.

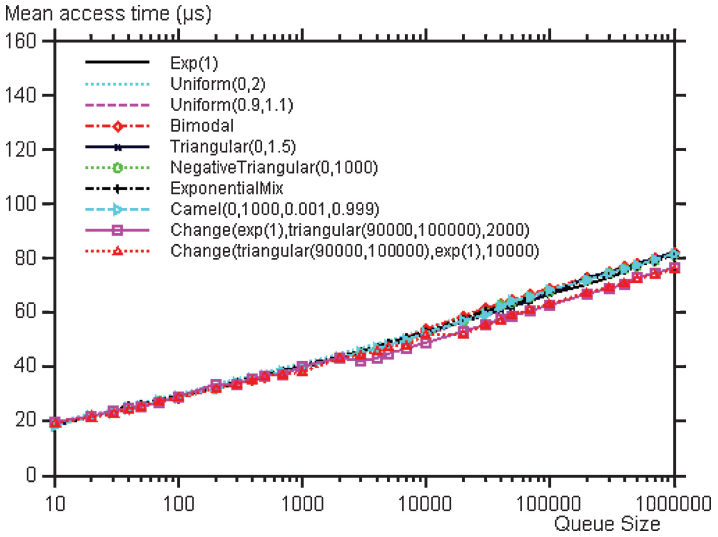

Fig. 11. Mean access time for splay tree under up/down experiments.

regardless of the queue size and priority increment distribution. This near O(1) performance corresponds very closely to the theoretical O(1) expected amortized performance as stated in Table III.

Amalgamated Splay Tree: Figure 4 shows that the amalgamated splay tree is near O(1) under the Classic Hold model. Figure 5 reveals that under the up/down model, for majority of the distributions, the Twol algorithm still offer near constant performance. However, this priority queue has lower performance

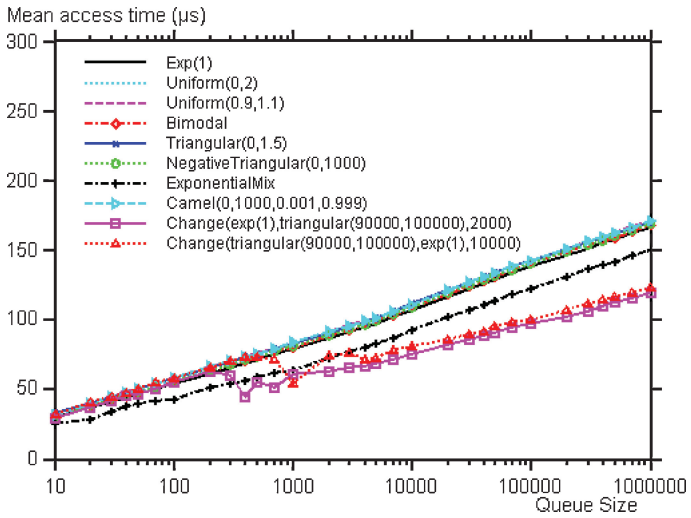

Fig. 12. Mean access time for skew heap under Classic Hold experiments.

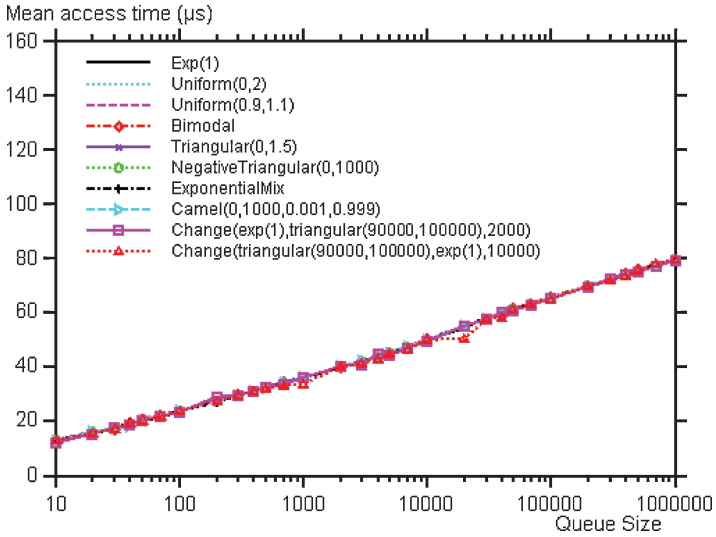

Fig. 13. Mean access time for skew heap under up/down experiments.

for some distributions where there are many events which fall into the basal structure, that is, the splay tree. It shows that the Twol algorithm is not adept at handling the exponentialMix, camel, and change distributions, where the means of the jump are varying, and relies on the basal structure to keep the access time bounded.

Amalgamated Skew Heap: Figures 6 and 7 display similar performance akin to that of the amalgamated splay tree.

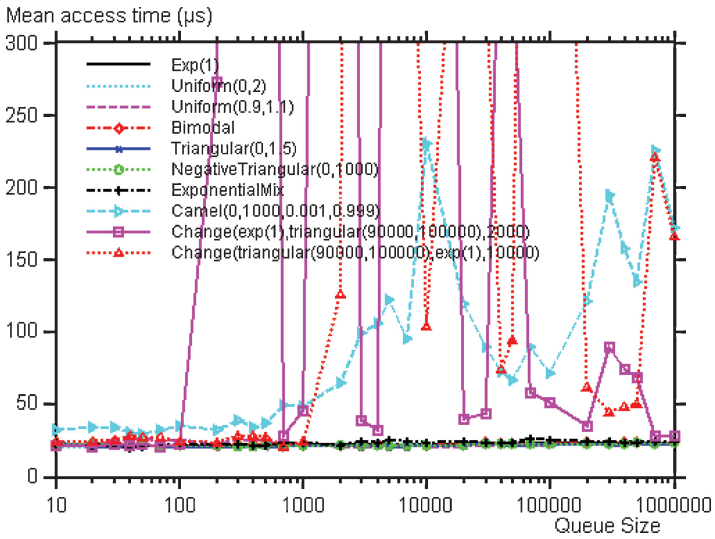

Fig. 14. Mean access time for calendar queue under Classic Hold experiments.

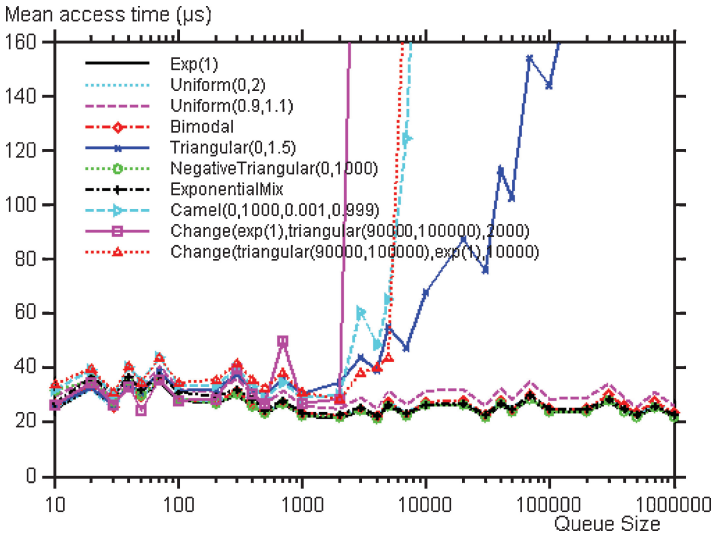

Fig. 15. Mean access time for calendar queue under up/down experiments.

These empirical results confirm, interestingly, that for both Classic Hold and up/down scenarios, if the mean of the jump μ does not vary, Twol-amalgamated priority queues exhibit near O(1) performance. This perhaps gives an insight to the possibility that perhaps Theorem 1 can be applicable even for varying N.

To make the performance improvement of employing the Twol algorithm more noticeable, Figures 8–13 present performance of the priority queues without amalgamation for comparison. Figures 14–17 demonstrate the performance of the popular CQ as well as its variant DCQ.

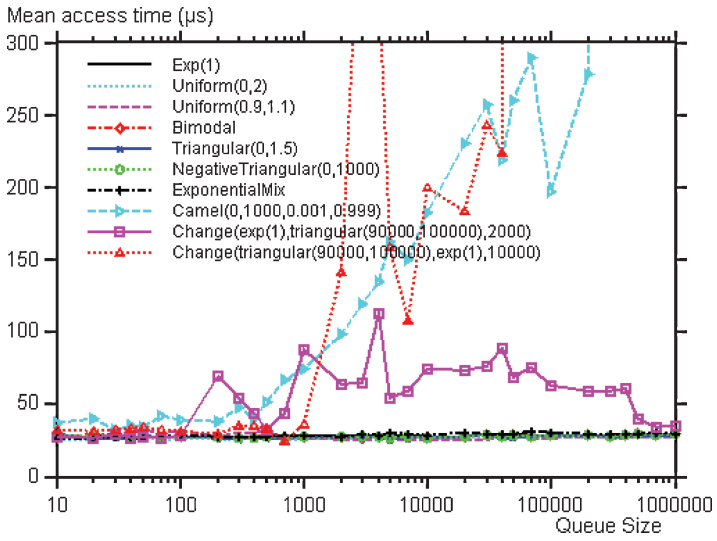

Fig. 16. Mean access time for dynamic calendar queue under Classic Hold experiments.

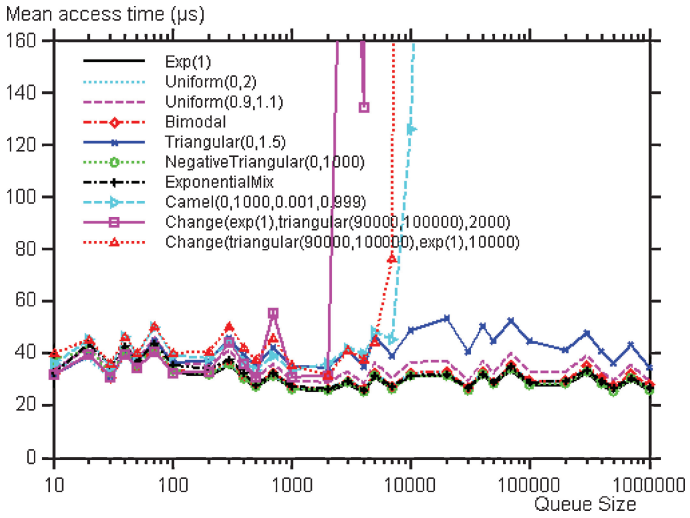

Fig. 17. Mean access time for dynamic calendar queue under up/down experiments.

Henriksen's Queue: Figure 8 empirically shows that the Henriksen's queue is nearly O(log(n)) for queue sizes up to a few thousand events. However for larger queue sizes, it appears to be very sensitive to the priority increment distribution. As a comparison with the amalgamated Henriksen, the Twol algorithm has very efficiently improved the performance of the Henriksen's queue. Figure 9 shows that the Henriksen's queue is efficient in managing widely varying queue sizes and is insensitive to the priority increment distribution, displaying a low O(log(n)) behavior.

Splay Tree: Figures 10 and 11 show the expected O(log(n)) behavior. It exhibits marginal sensitivity to priority increment distribution and this is perhaps due to its balancing heuristics. Generally, the splay tree performs worse than the Henriksen's queue.

Skew Heap: Figures 12 and 13 show that the access time grows at O(log(n)) and performs marginally better than the splay tree. The skew heap is a tad sensitive to priority increment distribution.

Calendar Queue: For most of the distributions in the Classic Hold model as shown in Figure 14, the CQ displays nearly constant performance, as expected. In the situations where its performance remains stable, it is apparent that the CQ outperforms all the other priority queues, including the Twol-amalgamated ones. It is precisely because of this efficiency that CQ is the default FEL candidate in the various discrete event simulators. However, the camel and change distributions demonstrate the inherent weakness of this popular priority queue. These distributions clearly show that if there are large number of events in a few buckets there would be sublists of O(n) length. Under a Classic Hold model, the ratio of events to buckets remains a constant and thus the heuristic is able to obtain a better bucketwidth, that is, the resize operation is not triggered. For the up/down model in Figure 15, the ratio of events to buckets changes frequently and this results in the resize operation being triggered often. This is graphically shown by the many peaks in the graphs. This access pattern model also shows that the CQ has poor performance for distributions such as the camel, change, and triangular.

Dynamic CQ: Figures 16 and 17 show a variant of the CQ which attempts to improve the heuristics for the resize criteria. There is performance improvement in managing events with the change(exp(1),triangular(90000, 100000), 2000) under the Classic Hold, and the triangular distribution under the up/down model. However this comes at a cost of slightly higher access time where its performance remains stable at near constant performance as compared to the CQ. These results suggest that its additional cost-based resize heuristics is generally considered ineffective.

Table VI and VII show the relative performance of the priority queues on the Intel Pentium 4. The CQ is the fastest for the exponential distribution in the Classic Hold model, while the amalgamated Henriksen is the overall leader when the priority increment distribution, queue size, and access pattern model are taken into account.

5.2 Performance on the Intel Itanium 2 (SGI Altix 3300), SGI MIPS R16000 (SGI Onyx4), and AMD Athlon MP

Due to the large number of figures and tables in this section, they have been mostly placed in the Appendix. Nevertheless Tables VIII and IX provide a vivid overall performance summary.

Table VIII presents in detail the actual speedup of the Twol algorithm amalgamation on all the architectures including the Intel Pentium 4 (with cache-disabled). It is conspicuous that the speedup is higher on the three cache-enabled architectures, with speedup ranging from about 3 to 5 times speed up

Table VI. Relative Performance for Exponential Distribution on Intel Pentium 4 (Normalized with Respect to the Fastest Access Time, Where the Higher the Number, the Slower It Is)

Model	Queue Size	Henriksen with Twol	Splay Tree with Twol	Skew Heap with Twol	Henriksen	Splay Tree	Skew Heap	CQ	DCQ
Classic Hold	10	1.28	2.05	1.49	1.04	2.36	1.39	**1.00**	1.29
	10^2	1.35	2.20	1.67	1.43	3.63	2.48	**1.00**	1.26
	10^3	1.36	2.29	1.75	1.84	4.75	3.58	**1.00**	1.26
	10^4	1.40	2.46	1.92	2.34	6.04	4.87	**1.00**	1.25
	10^5	1.41	2.63	2.14	2.72	6.95	5.92	**1.00**	1.23
	10^6	1.40	2.57	2.15	3.14	7.98	7.03	**1.00**	1.23
	Avg.	1.37	2.37	1.85	2.09	5.29	4.21	**1.00**	1.25
Up/Down	10	1.46	1.87	1.51	**1.00**	1.70	1.23	2.30	2.97
	10^2	1.18	1.56	1.25	**1.00**	2.03	1.68	1.96	2.31
	10^3	**1.00**	1.40	1.15	1.14	2.35	2.13	1.30	1.54
	10^4	**1.00**	1.43	1.16	1.41	3.00	2.86	1.50	1.79
	10^5	**1.00**	1.44	1.17	1.67	3.67	3.60	1.31	1.54
	10^6	**1.00**	1.52	1.46	1.96	4.42	4.31	1.17	1.39
	Avg.	**1.11**	1.54	1.28	1.36	2.86	2.64	1.59	1.92
Total Avg.		**1.24**	1.95	1.57	1.72	4.07	3.42	1.30	1.59

Table VII. Relative Average Performance for All Distributions on Intel Pentium 4 (Normalized with Respect to the Fastest Access Time, where the Higher the Number, the Slower It Is)

Model	Queue Size	Henriksen with Twol	Splay Tree with Twol	Skew Heap with Twol	Henriksen	Splay Tree	Skew Heap	CQ	DCQ
Classic Hold	10	**1.00**	5.03	3.91	2.50	6.40	3.72	2.78	3.47
	10^2	1.36	2.08	1.58	1.23	3.18	2.32	**1.00**	1.23
	10^3	1.24	1.97	1.52	1.48	3.41	2.77	**1.00**	1.43
	10^4	**1.00**	1.68	1.32	1.68	3.43	2.99	6.06	1.91
	10^5	**1.00**	1.58	1.31	2.47	4.11	3.75	3.19	NA
	10^6	**1.00**	1.57	1.34	3.74	4.61	4.32	1.45	NA
	Avg.	**1.10**	2.32	1.83	2.18	4.19	3.31	2.58	NA
Up/Down	10	1.49	1.85	1.52	**1.00**	1.76	1.18	2.65	3.20
	10^2	1.21	1.74	1.53	**1.00**	2.04	1.67	2.14	2.49
	10^3	**1.00**	1.52	1.34	1.07	2.25	2.01	1.44	1.68
	10^4	**1.00**	1.61	1.48	1.23	2.66	2.54	14.63	14.25
	10^5	**1.00**	1.63	1.43	1.46	3.22	3.15	NA	NA
	10^6	**1.00**	1.66	1.47	1.71	3.90	3.85	NA	NA
	Avg.	**1.12**	1.67	1.46	1.25	2.64	2.40	NA	NA
Total Avg.		**1.11**	1.99	1.65	1.71	3.41	2.86	NA	NA

over the basal structure without amalgamation. For the Intel Pentium 4 with cache disabled, the speedup is only about 2. Even though the cache-disabled experiments on Intel Pentium 4 are done to illustrate a closer approximation of the complexity of the priority queues and not meant to reflect the expected speedup of normal applications, we have nevertheless included it to obtain the overall average speedup of 3.23.

Table VIII. Speedup[a] of Twol Algorithm on Different Hardware Architecture—Comparison by Priority Increment Distribution (if the Values Exceed 1, it Means There is a Speedup)

Priority Queue	Amalgamated Henriksen				Amalgamated Splay Tree				Amalgamated Skew Heap				
Architecture	P4[b]	IT2	MIPS	AMD	P4[b]	IT2	MIPS	AMD	P4[b]	IT2	MIPS	AMD	Avg.
Exponential(1)	1.46	3.12	2.57	3.31	2.20	4.13	3.58	4.04	2.32	4.79	3.24	3.43	3.18
Uniform(0, 2)	2.28	9.83	6.10	7.03	2.78	3.82	3.84	4.01	2.76	4.46	3.42	3.41	4.48
Uniform(0.9, 1.1)	1.65	4.90	3.36	3.71	2.43	3.02	3.39	3.06	3.03	4.50	3.68	3.52	3.35
Bimodal	1.61	5.86	3.93	4.73	2.53	3.95	3.65	3.99	2.60	4.65	3.45	3.44	3.70
Triangular (0, 1.5)	2.08	10.65	6.57	7.42	2.60	3.63	3.74	3.77	2.90	4.47	3.55	3.45	4.57
NegativeTriangular (0, 1000)	1.84	7.20	4.70	5.55	2.52	3.95	3.79	4.05	2.74	4.56	3.43	3.40	3.98
ExponentialMix	1.17	2.56	1.88	2.71	1.25	2.54	2.06	2.75	1.24	2.85	1.92	2.45	2.12
Camel (0, 1000, 0.001, 0.999)	1.10	3.03	1.85	2.37	1.31	2.52	1.91	2.21	1.34	4.01	2.26	2.89	2.23
Change(exp(1), triangular (90000, 100000), 2000)	1.43	4.11	2.58	2.87	1.54	2.20	1.98	2.03	1.64	3.14	2.26	2.29	2.34
Change(triangular (90000, 100000), exp(1), 10000)	1.38	4.06	2.60	2.84	1.50	2.17	2.01	1.94	1.76	3.16	2.28	2.30	2.33
Average	1.60	5.53	3.61	4.25	2.07	3.19	2.99	3.19	2.23	4.06	2.95	3.06	**3.23**

[a]Speedup refers to the performance of the amalgamated version normalized over its nonamalgamated version.
[b]P4 is with cache disabled.

Table IX gives an itemization of the average speedup as the queue size varies from 10 to 10 million events. The average speedup of the amalgamated priority queues over its nonamalgamated form is 2.80 across different basal structures and different hardware architectures.

Note that for Tables VIII and IX, the range of queue sizes tested for P4 (Intel Pentium4) is from 10 to 1 million, IT2 (Intel Itanium 2) is from 100 to 10 million, and MIPS (SGI MIPS R16000) and AMD (AMD Athlon MP) are from 100 to 1 million as mentioned in Table V. The IT2, MIPS, and AMD start from queue size 100 because the efficient cache on these architectures enables the benchmark to run so fast that we could not obtain accurate timings for queue sizes below 100, due to the limitation of the microsecond timer. We have benchmarked up to 10 million events only on the IT2, which is fast and has ample shared RAM.

5.3 Cost versus Performance Consideration

After numerous verifications to validate the performance of the Twol-amalgamated priority queues, this section serves to discuss the additional cost in terms of additional memory required for the Twol algorithm to function. The bulk of memory requirement in the Twol structure is in its first tier (T_1) where the number of buckets allocated is bounded above by $N + 1$ (see Lemma 5). Other memory requirements are some variables associated with T_1 and T_2 as mentioned in Section 2 which can be considered to take up an insignificant amount of memory as compared to the bucket allocation.

The structure of a bucket in C programming language is given as `struct bucket {struct event *head; struct event *tail;}`, which incurs a cost of

Table IX. Speedup of Twol Algorithm on Different Hardware Architecture—Comparison by Queue Size (If the Values Exceed 1, it Means There Is a Speedup)

Priority Queue	Amalgamated Henriksen				Amalgamated Splay Tree				Amalgamated Skew Heap				Avg	Cost (MB)[a]
Architecture Queue Size	P4+	IT2	MIPS	AMD	P4+	IT2	MIPS	AMD	P4+	IT2	MIPS	AMD		
10	0.73	Inaccurate timings			1.17	Inaccurate timings			0.89	Inaccurate timings			0.93	1.76e-04
20	0.76	due to the efficient			1.25	due to the efficient			1.05	due to the efficient			1.02	3.36e-04
30	0.78	cache and the			1.25	cache and the			1.13	cache and the			1.05	4.96e-04
40	0.81	limitation of the			1.35	limitation of the			1.21	limitation of the			1.12	6.56e-04
50	0.85	microsecond			1.31	microsecond			1.20	microsecond			1.12	8.16e-04
70	0.86	timer.			1.37	timer.			1.28	timer.			1.17	1.14e-03
100	0.91	0.88	1.03	0.87	1.43	1.42	1.59	1.40	1.34	1.19	1.31	1.18	1.21	1.62e-03
200	0.97	0.96	1.13	0.93	1.51	1.52	1.66	1.55	1.44	1.35	1.40	1.33	1.31	3.22e-03
300	1.01	1.02	1.19	1.00	1.56	1.58	1.70	1.57	1.55	1.43	1.47	1.36	1.37	4.82e-03
400	1.04	1.07	1.23	1.00	1.60	1.62	1.74	1.62	1.58	1.47	1.51	1.39	1.41	6.42e-03
500	1.07	1.10	1.27	1.01	1.61	1.64	1.74	1.58	1.64	1.50	1.52	1.39	1.42	8.02e-03
700	1.11	1.16	1.33	1.05	1.64	1.69	1.78	1.59	1.68	1.55	1.57	1.42	1.46	1.12e-02
1000	1.15	1.21	1.40	1.13	1.65	1.74	1.84	1.65	1.71	1.59	1.63	1.46	1.51	1.60e-02
2000	1.24	1.34	1.51	1.28	1.69	1.84	1.92	1.79	1.78	1.70	1.72	1.54	1.61	3.20e-02
3000	1.31	1.44	1.61	1.42	1.76	1.99	2.00	1.88	1.86	1.82	1.82	1.68	1.72	4.80e-02
4000	1.34	1.50	1.65	1.55	1.78	2.03	2.03	1.94	1.89	1.90	1.87	1.71	1.77	6.40e-02
5000	1.38	1.56	1.70	1.64	1.81	2.07	2.07	1.99	1.94	1.95	1.92	1.77	1.82	8.00e-02
7000	1.45	1.64	1.77	1.95	1.86	2.11	2.12	2.04	2.00	2.04	1.98	1.80	1.90	1.12e-01
10000	1.52	1.70	1.84	2.21	1.90	2.15	2.17	2.05	2.05	2.12	2.04	1.78	1.96	1.60e-01
20000	1.67	1.90	2.04	2.72	2.03	2.28	2.32	2.36	2.21	2.34	2.20	2.15	2.19	3.20e-01
30000	1.75	2.10	2.23	3.02	2.10	2.43	2.42	2.56	2.30	2.47	2.24	2.39	2.33	4.80e-01
40000	1.85	2.57	2.50	3.35	2.14	2.72	2.61	2.73	2.38	2.57	2.39	2.58	2.53	6.40e-01
50000	1.88	2.87	2.77	3.52	2.19	2.76	2.79	2.85	2.45	2.67	2.53	2.73	2.67	8.00e-01
70000	1.98	3.37	3.38	3.86	2.25	2.72	3.19	3.03	2.54	2.80	2.87	2.95	2.91	1.12
100000	2.09	3.63	3.58	4.24	2.36	2.67	3.27	3.24	2.60	2.99	3.09	3.17	3.08	1.60
200000	2.34	4.09	4.37	4.99	2.47	2.76	3.44	3.64	2.72	3.46	3.43	3.56	3.44	3.20
300000	2.50	4.46	4.79	5.52	2.58	2.87	3.56	3.87	2.79	3.71	3.67	3.80	3.68	4.80
400000	2.59	4.78	5.13	5.98	2.62	2.97	3.67	4.07	2.90	3.89	3.81	3.97	3.87	6.40
500000	2.67	5.01	5.37	6.33	2.66	3.07	3.75	4.17	2.94	4.06	3.93	4.12	4.01	8.00
700000	2.86	5.46	5.86	6.93	2.72	3.23	3.89	4.36	3.00	4.30	4.12	4.35	4.26	11.20
1000000	3.01	5.96	6.38	7.62	2.76	3.41	4.06	4.59	3.05	4.56	4.31	4.62	4.53	16.00
2000000	—	7.03	—	—	—	3.73	—	—	—	5.07	—	—	5.28	32.00
3000000	—	7.93	—	—	—	3.93	—	—	—	5.35	—	—	5.74	48.00
4000000	—	8.49	—	—	—	4.04	—	—	—	5.52	—	—	6.02	64.00
5000000	—	8.99	—	—	—	4.14	—	—	—	5.65	—	—	6.26	80.00
7000000	—	9.83	—	—	—	4.35	—	—	—	5.89	—	—	6.69	112.00
10000000	—	10.73	—	—	—	4.51	—	—	—	6.09	—	—	7.11	160.00
Average	1.53	3.73	2.68	3.00	1.88	2.64	2.53	2.56	1.97	3.06	2.41	2.41	**2.80**	—

[a]Assume a bucket allocated in T_1 of Twol algorithm requires 16 bytes of shared RAM. The cost is an upper bound for individual queue sizes. MB here refers to 1,000,000 bytes.

8 B (or 8 bytes) on 32-bit and up to 16 B on 64-bit hardware platform. Assuming the cost of 16 B per bucket allocated, the total cost and the speedup at corresponding queue sizes on different platforms are given in Table IX. At 1 million queue size, the average speedup is 4.53 and the maximum memory requirement is 16 MB, which by today's workstation or server configuration is considered

nominal. Furthermore low cost 64-bit processors such as the AMD Opteron and the AMD Athlon64 (a PC processor which is likely to become a desktop commodity in the near future) have recently been introduced. Generally, a 32-bit processor allows memory addressing of only up to 4 GB whereas a 64-bit processor theoretically allows memory addressing of up to 16 exabytes (16 billion GB) and thus it is limited only by hardware implementation. Even though some operating systems such as Linux interestingly enables 64 GB physical memory addressing on 32-bit processors, the individual applications and the operating system still cannot access more than 4 GB each. With the advent of 64-bit computing, the minor drawback of the Twol algorithm, that is, to trade a small amount of memory for speed, seems trivial. From Table IX, the speedup increases as queue size increases and thus shows that it is especially suitable for implementation in large-scale applications.

Under stringent circumstance where there is memory scarcity, the Twol algorithm is still able to function by allocating the number of buckets which is less than the queue size. We can assume that there are n_x events on the average in each bucket, contrary to the one event per bucket in Eq. (1), and thus the number of buckets that the Twol algorithm requires is now n_x^{-1} of the queue size. We then obtain

$$\text{Bucketwidth of } T_1 = T_{1\_Bw} = n_x \cdot (T_{2\_Max} - T_{2\_Min}/T_{2\_Num}), \tag{9}$$

which means that the bucketwidth is directly proportional to n_x. The time interval that spans one year in T_1 remains the same at $T_1\_Bw \cdot$ (number of buckets). So, for instance, if we assume two events in each bucket during a transfer of N events from T_2 to T_1, we would require only $(N/2) + 1$ number of buckets, that is, $\sim$ half the memory required, than if we assume one event per bucket. As $n_x \rightarrow T_2\_num$, the performance of the amalgamated priority queue tends toward that of the basal structure. Since the expected number of events in each bucket will contain more events as the number of buckets decreases, it is therefore expected to have gradual performance degradation until $n_x = T_2\_num$, where the Twol-amalgamated priority queue performs similarly to its basal structure.

5.4 Performance Evaluation via Swan on Intel Pentium 4 (Cache Enabled)

To determine the performance of the priority queues in real-discrete event simulation, the CQ, skew heap, splay tree, Henriksen's queue, and their Twol-amalgamated counterparts, have been implemented as the FEL structures available in the *Swan* (simulator without a name) simulator [Thng and Goh 2004]. A simple $M/M/1$ queuing system was created in Swan and simulated for 10,000 simulation seconds. In this network topology, each source node generates a network packet where the interpacket generation rate is exponentially distributed. The source nodes, which vary from 8 to 399,998, generate packets which are multiplexed into a single server that services the packets to a single destination node. The service times for the packets are also exponentially distributed. The results obtained are shown in Figures 18 and 19.

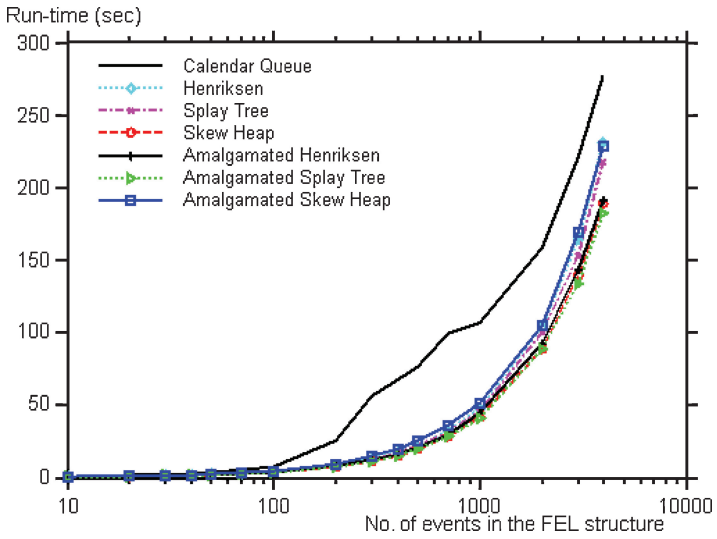

Fig. 18. Run-time performance measurements in Swan on Intel Pentium 4 for 10 to 4,000 network nodes.

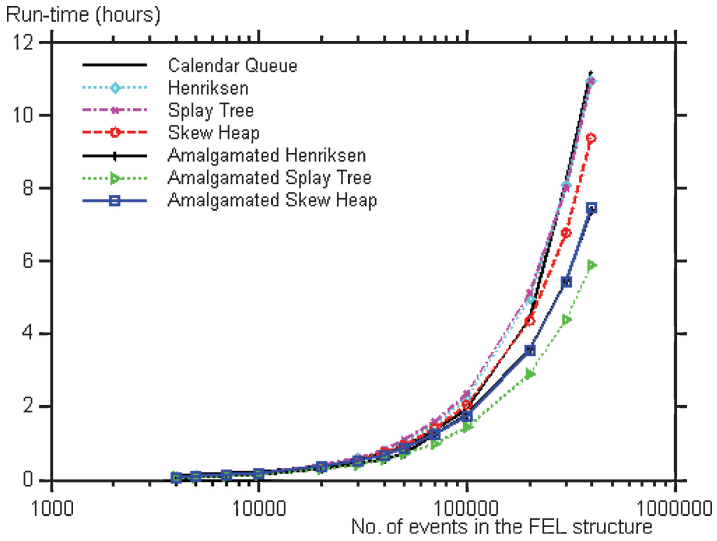

Fig. 19. Run-time performance measurements in Swan on Intel Pentium 4 for 4,000 to 400,000 network nodes.

Each discrete event simulator operates uniquely. For Swan, an event in the FEL corresponds to a node in the network topology. At the onset, if the number of nodes is set to be N, Swan initializes the FEL structure with N events with each holding a zero timestamp. Thereafter, the process interactions between the nodes determine the DeleteMin and Insert operation of subsequent events with the number of events in the FEL structure being held constant throughout

the simulation at N. Thus the mechanism of Swan essentially corresponds to an initial up access pattern model (see Section 4.1). This creates N number of nodes and then the simulation follows a Classic Hold model where the number of events in the FEL structure is kept constant. The number of events in the FEL is equal to the number of nodes in the network topology. Thus the x-axis in Figures 18 and 19 can be relabeled as "number of nodes."

The benchmarks in this section were carried out on the Intel Pentium 4 workstation as described in Table V with both L1 and L2 cache enabled. In addition, the operating system used is Windows XP Professional SP 1. Unlike in previous benchmarks where the caches were turned off to illustrate the performance of the priority queues without cache effects, this set of benchmarks using the Swan simulator is carried out to demonstrate the actual integral role that a high performance priority queue assumes during a discrete event simulation. Intel Pentium 4 was chosen because the Swan simulator is currently available only on Windows platform (although with minor tweaks, it can be made to run on Linux or other platforms since it is written in ANSI C++) and the fastest compatible hardware platform was chosen to reduce the run-time taken. The metric measured was the overall run-time, which includes the Swan simulation engine management and control time, in addition to the FEL structure management time. So while the FEL structure may have constant complexity, the overall run-time is not expected to be constant. Nevertheless, the results illustrate a realistic view on the degree to which the performance of an FEL structure can affect the run-time. The run-time for each simulation with different FEL structure corresponds to taking the average value from five identical runs.

Figure 18 show that for relatively small number of events up to a few thousand, the cache memory of the hardware brings about good performance for almost all priority queues except for the CQ. This is because due to the initial up access pattern model, the CQ resizes frequently and this results in a poorer performance. However if the simulation time is much larger, the CQ is likely to have comparable result since the run-time of the up model will be less significant. The amalgamated splay tree, amalgamated Henriksen, and skew heap have almost equivalent performance. Interestingly, the amalgamated skew heap is slower than its nonamalgamated version. Figure 19 shows that for relatively large-scale topology, the runtime can vary from 6 to 11 h. The amalgamated priority queues come out tops, with the amalgamated splay tree leading the pack. It posts nearly a 100% speedup at 400,000 queue size.

To this end, the Twol algorithm has again shown to be a conspicuously efficient structure for amalgamation with priority queues particularly for large-scale simulation scenarios.

6. CONCLUSION

The choice of an efficient priority queue structure plays an integral role in numerous applications especially for sizeable application scenarios. Tree-based

priority queues offer stable performance regardless of priority increment distribution but they are limited by O(log(n)) amortized complexity. On the other hand, the calendar queue (CQ) is a list-based structure that exhibits near constant performance under some simulation scenarios, which makes it a popular implementation in simulators. However, the CQ performs poorly under other scenarios in which the mean of the jump of the event distribution varies, such as skewed distributions in which many events fall into a few buckets in the CQ. This paper proposes a new implementation called the Twol structure that amalgamates with three efficient priority queues, the Henriksen's queue, skew heap, and splay tree, to achieve very efficient priority queues. Twol-amalgamated priority queues not only exhibit the stability of the basal structures but also the O(1) characteristic found in the list-based CQ. To achieve these properties, the Twol algorithm employs a two-tier list-based structure where the first-tier spreads out the events while the second acts as an overflow structure. Our theoretical studies demonstrate that a Twol-amalgamated priority queue that maintains N active events offers O(1) expected amortized complexity under the assumption that the mean of the jump of the priority increment distribution remains a constant. Extensive empirical studies on the performance of the amalgamated priority queues on different hardware platforms, in comparison with the basal priority queues and other popular implementations, are also presented. These empirical studies confirm the theoretical predictions that the amalgamated priority queues have indeed near constant performance and are bounded by the basal structure's amortized complexity. Furthermore, the Twol structure offers three to five times speedup for the three basal structures considered. The stable and efficient near O(1) growth performance under all priority increment distributions for the Classic Hold and up/down experiments, for queue sizes ranging from 10 to 10 million events, demonstrates that it is the superior structure for small to large-scale discrete event simulation.

APPENDIX

Figures 20 and 25 demonstrate the extent to which different hardware architectures and compiler code generation affect the performance of the various priority queues. The "knees" in these figures are due to declining cache performance. For the cache-rich Intel Itanium 2 and SGI MIPS R16000 architectures, the knee-effect occurs at queue size of about 30,000 while for the AMD, in the neighborhood of 6,000. It is apparent that the priority queues have similar performance curves on these three widely different hardware platforms.

Tables X to XV show explicitly the relative performance comparison between the eight priority queues. Essentially, in all the three architectures, the tables conclusively show that the best performing priority queues are those which are amalgamated with the Twol algorithm. In particular, on Intel Itanium 2, the amalgamated skew heap comes out tops, whereas for SGI MIPS R16000 and AMD Athlon MP, the amalgamated Henriksen's queue emerges as the fastest priority queue.

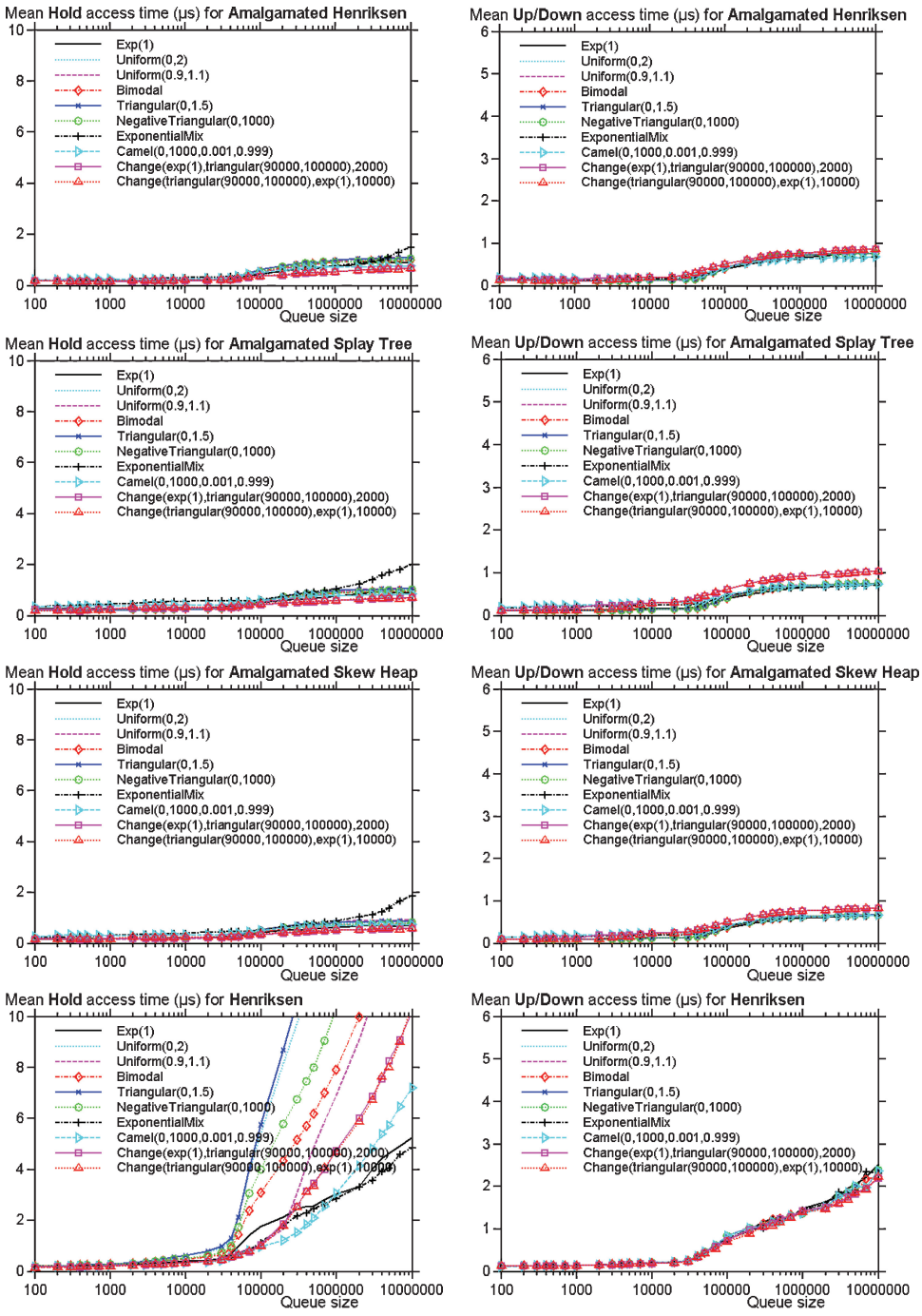

Fig. 20. Performance measurements on Intel Itanium 2 (SGI Altix 3300).

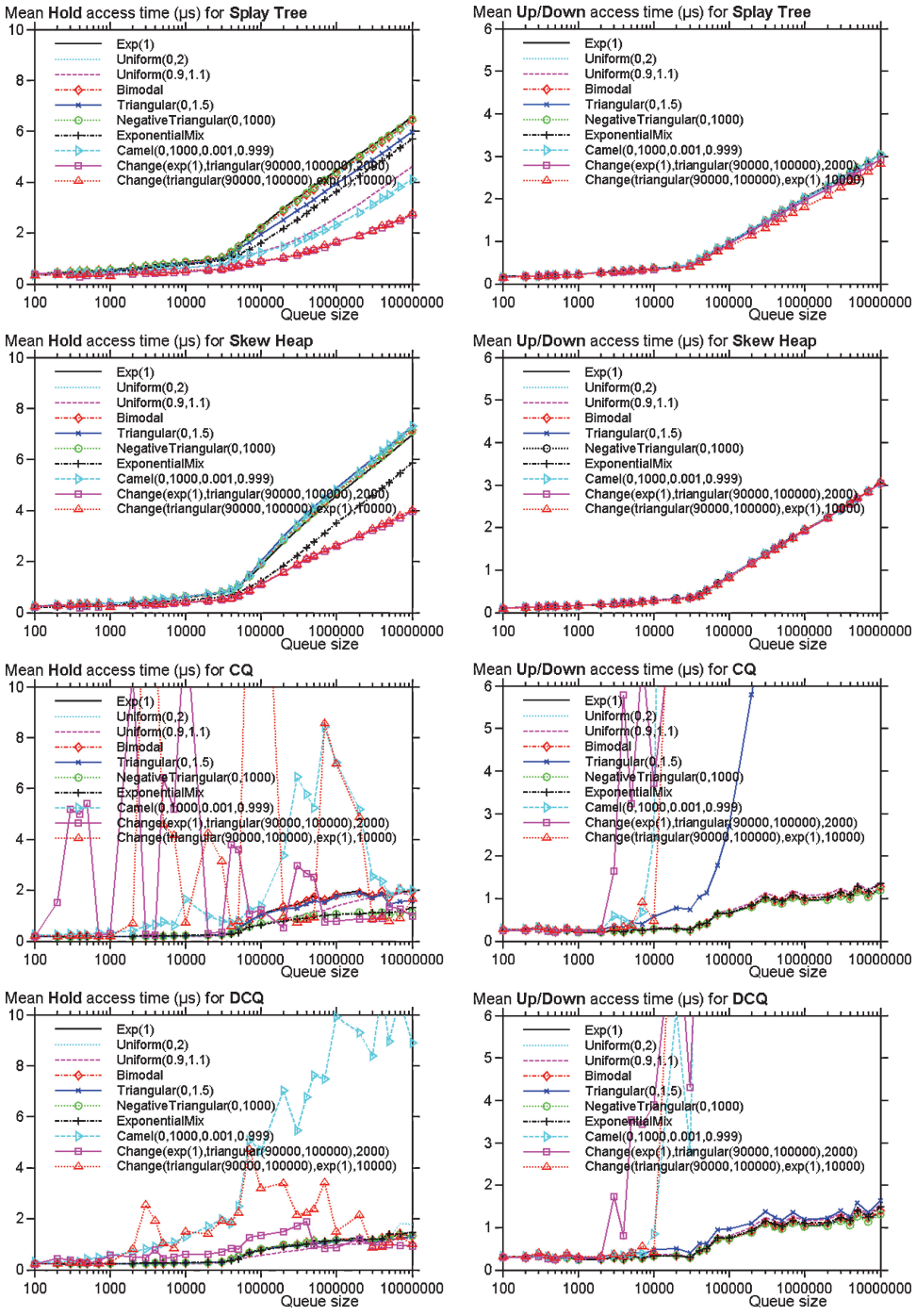

Fig. 21. Performance measurements on Intel Itanium 2 (SGI Altix 3300).

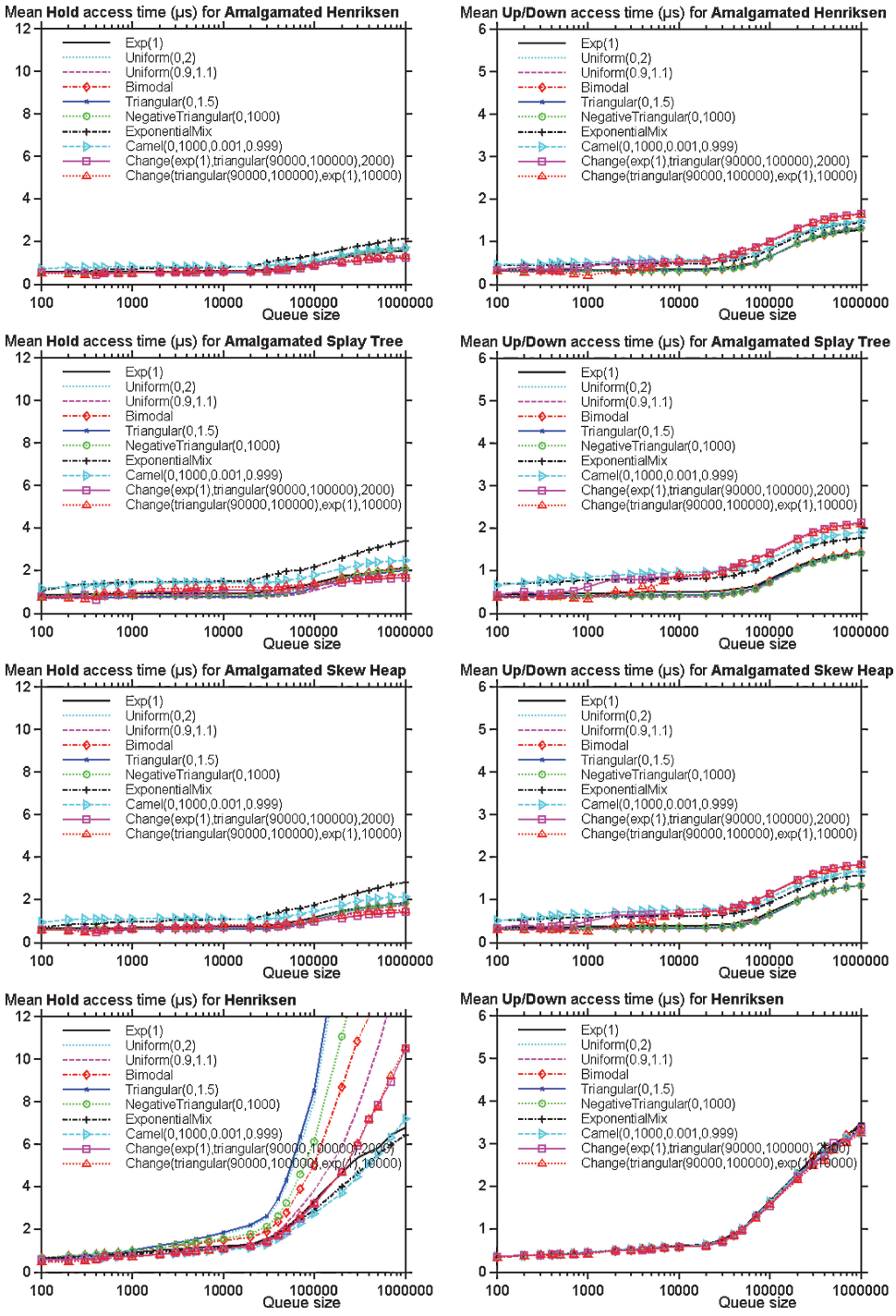

Fig. 22. Performance measurements on SGI MIPS R16000 (SGI Onyx4).

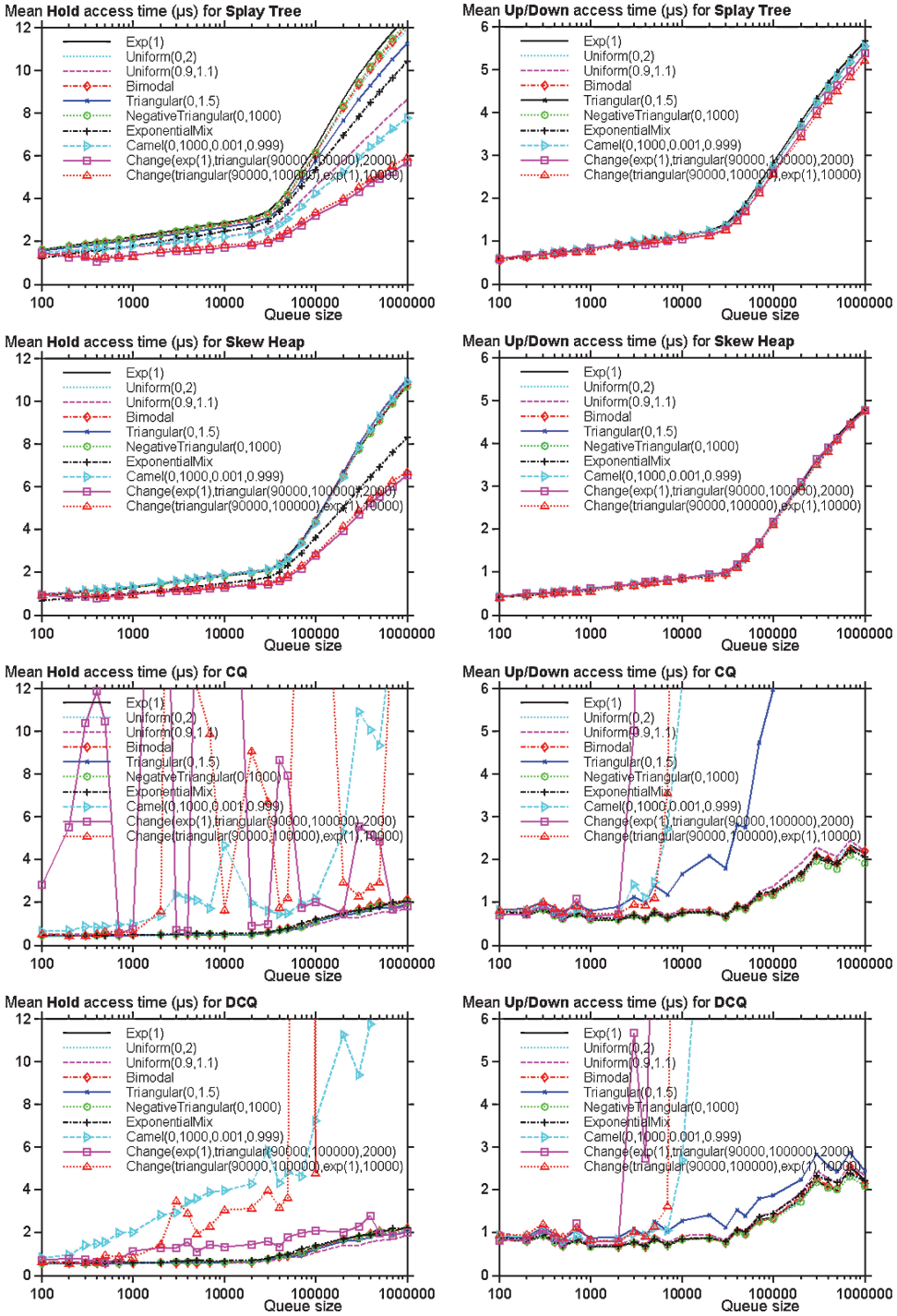

Fig. 23. Performance measurements on SGI MIPS R16000 (SGI Onyx4).

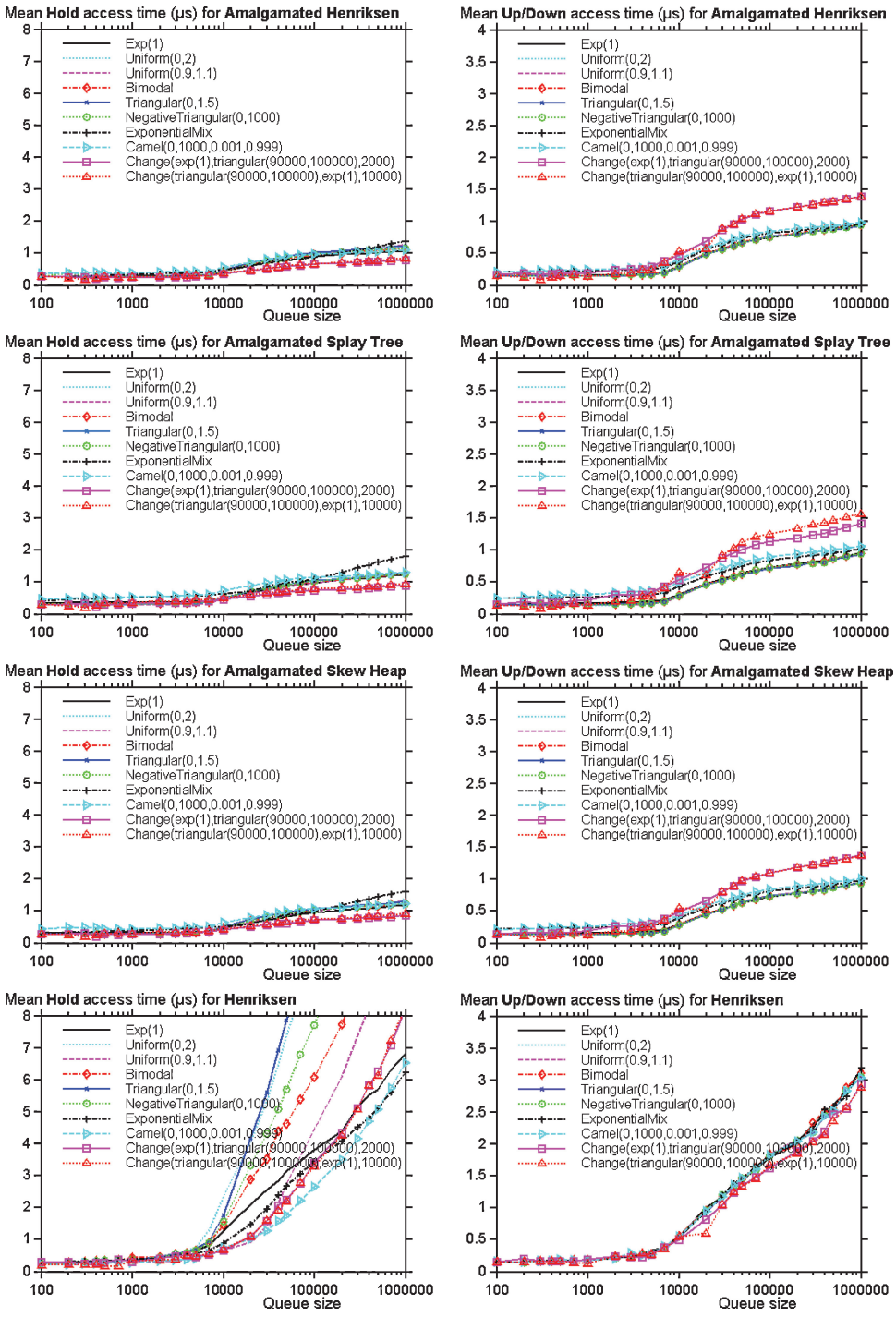

Fig. 24. Performance measurements on AMD Athlon MP.

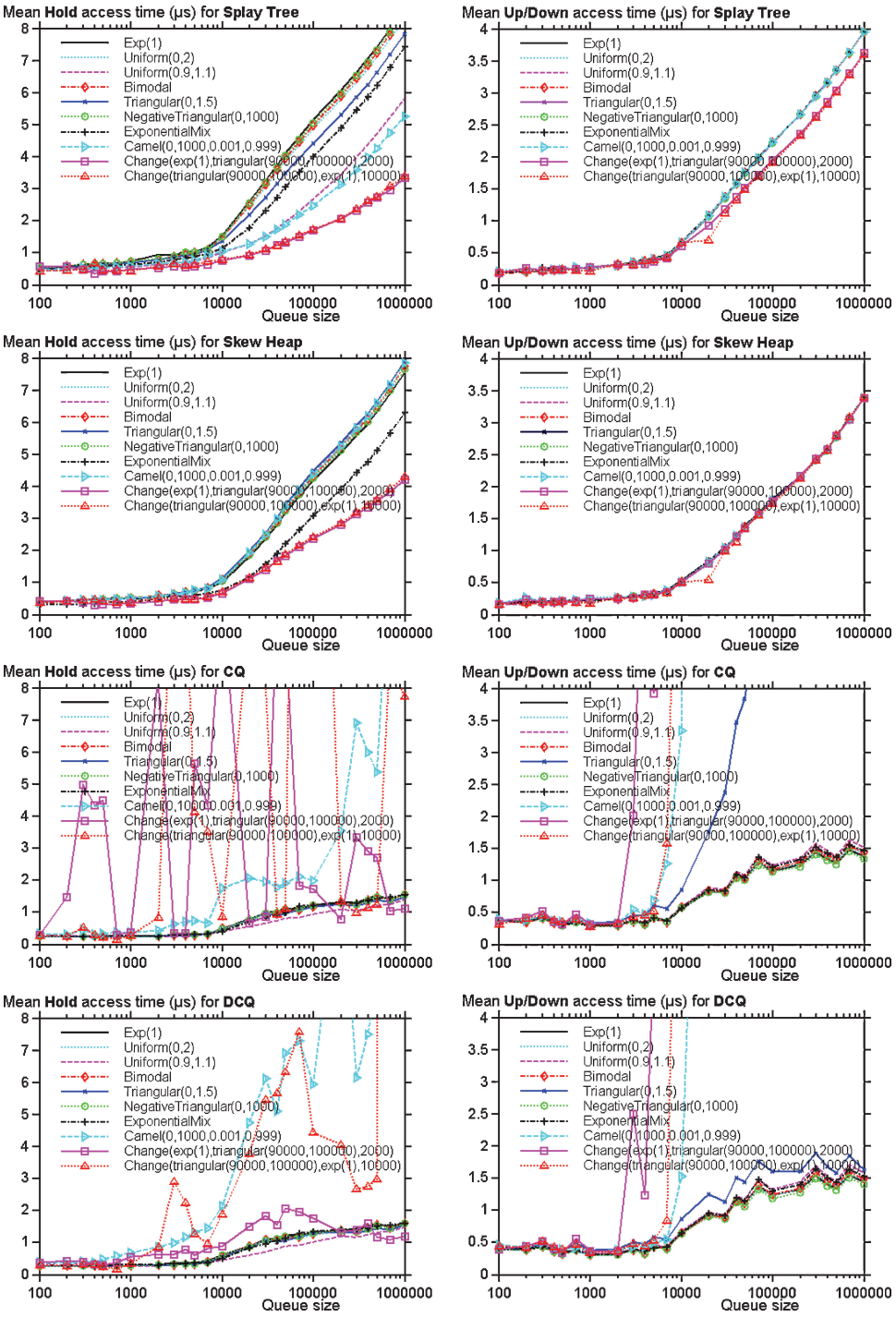

Fig. 25. Performance measurements on AMD Athlon MP.

Table X. Relative Performance for Exponential Distribution on Intel Itanium 2 (Normalized with Respect to the Fastest Access Time, Where the Higher the Number, the Slower It Is)

Model	Queue Size	Henriksen with Twol	Splay Tree with Twol	Skew Heap with Twol	Henriksen	Splay Tree	Skew Heap	CQ	DCQ
Classic Hold	10	1.09	1.43	**1.00**	1.05	2.31	1.36	1.01	1.38
	10^2	1.01	1.47	1.04	1.38	3.17	1.92	**1.00**	1.34
	10^3	**1.00**	1.50	1.10	1.90	4.33	2.98	1.03	1.33
	10^4	1.08	1.22	**1.00**	4.26	5.52	4.51	2.54	1.97
	10^5	1.19	1.23	**1.00**	4.96	7.32	7.51	3.00	1.92
	10^6	1.28	1.31	**1.00**	7.64	9.56	10.14	2.88	1.93
	Avg.	1.11	1.36	**1.02**	3.53	5.37	4.74	1.91	1.65
Up/Down	10	1.58	1.28	**1.00**	1.40	1.69	1.10	2.82	3.28
	10^2	1.18	1.32	**1.00**	1.39	2.34	1.69	2.13	2.52
	10^3	1.04	1.23	**1.00**	1.53	2.90	2.23	2.26	2.60
	10^4	1.11	1.10	**1.00**	2.14	2.83	2.48	1.87	2.14
	10^5	1.13	1.10	**1.00**	2.52	3.46	3.31	1.63	1.78
	10^6	1.17	1.14	**1.00**	3.87	4.69	4.72	1.89	2.11
	Avg.	1.20	1.20	**1.00**	2.14	2.99	2.59	2.10	2.41
Total Avg.		1.16	1.28	**1.01**	2.84	4.18	3.66	2.01	2.03

Table XI. Relative Average Performance for All Distributions on Intel Itanium 2 (Normalized with Respect to the Fastest Access Time, Where the Higher the Number, the Slower It Is)

Model	Queue Size	Henriksen with Twol	Splay Tree with Twol	Skew Heap with Twol	Henriksen	Splay Tree	Skew Heap	CQ	DCQ
Classic Hold	10	1.11	1.48	1.09	**1.00**	2.23	1.41	1.13	1.47
	10^2	**1.00**	1.47	1.07	1.30	2.62	1.78	1.13	1.70
	10^3	**1.00**	1.46	1.11	1.99	3.37	2.62	7.83	2.47
	10^4	1.07	1.19	**1.00**	5.65	3.72	3.80	6.53	3.26
	10^5	1.13	1.19	**1.00**	10.78	4.76	6.02	3.64	2.88
	10^6	1.16	1.23	**1.00**	19.54	6.22	7.72	1.95	2.47
	Avg.	1.07	1.33	**1.05**	6.61	3.79	3.85	3.69	2.36
Up/Down	10	1.46	1.24	**1.00**	1.24	1.54	1.01	2.64	3.06
	10^2	1.09	1.24	**1.00**	1.18	2.03	1.47	1.99	2.36
	10^3	**1.00**	1.27	1.06	1.29	2.4	1.86	5.94	5.09
	10^4	1.06	1.12	**1.00**	1.92	2.38	2.09	NA	NA
	10^5	1.09	1.13	**1.00**	2.14	3.07	2.98	NA	NA
	10^6	1.1	1.14	**1.00**	3.35	4.33	4.38	NA	NA
	Avg.	1.13	1.19	**1.01**	1.85	2.63	2.30	NA	NA
Total Avg.		1.11	1.26	**1.03**	4.28	3.22	3.10	NA	NA

Table XII. Relative Performance for Exponential Distribution on SGI MIPS R16000 (Normalized with Respect to the Fastest Access Time, Where the Higher the Number, the Slower It Is)

Model	Queue Size	Henriksen with Twol	Splay Tree with Twol	Skew Heap with Twol	Henriksen	Splay Tree	Skew Heap	CQ	DCQ
Classic Hold	10^2	1.33	1.98	1.47	1.48	3.75	2.09	**1.00**	1.34
	10^3	1.30	2.01	1.51	2.01	4.85	2.84	**1.00**	1.28
	10^4	1.33	2.05	1.57	2.60	6.19	3.92	**1.00**	1.32
	10^5	**1.00**	1.45	1.19	3.41	6.56	4.57	1.15	1.37
	10^6	**1.00**	1.37	1.19	4.35	8.11	6.90	1.32	1.43
	Avg.	1.19	1.77	1.39	2.77	5.89	4.06	**1.09**	1.35
Up/Down	10^2	1.01	1.30	**1.00**	1.05	1.72	1.23	2.11	2.47
	10^3	**1.00**	1.47	1.13	1.41	2.57	1.85	1.80	2.09
	10^4	**1.00**	1.46	1.14	1.74	3.31	2.46	2.15	2.45
	10^5	**1.00**	1.24	1.08	2.58	4.42	3.39	1.87	2.11
	10^6	**1.00**	1.13	1.04	2.72	4.43	3.78	1.67	1.75
	Avg.	**1.00**	1.32	1.08	1.90	3.29	2.54	1.92	2.17
Total Avg.		**1.10**	1.55	1.23	2.34	4.59	3.30	1.51	1.76

Table XIII. Relative Average Performance for All Distributions on SGI MIPS R16000 (Normalized with Respect to the Fastest Access Time, Where the Higher the Number, the Slower It Is)

Model	Queue Size	Henriksen with Twol	Splay Tree with Twol	Skew Heap with Twol	Henriksen	Splay Tree	Skew Heap	CQ	DCQ
Classic Hold	10^2	**1.00**	1.49	1.15	1.04	2.62	1.59	1.26	1.09
	10^3	1.08	1.73	1.28	1.57	3.37	2.19	**1.00**	1.46
	10^4	**1.00**	1.62	1.21	2.10	3.84	2.70	6.55	1.99
	10^5	**1.00**	1.42	1.16	4.65	5.08	3.96	4.17	2.28
	10^6	**1.00**	1.31	1.15	9.84	5.99	5.93	2.49	NA
	Avg.	**1.02**	1.51	1.19	3.84	4.18	3.27	3.09	NA
Up/Down	10^2	**1.00**	1.29	1.01	1.00	1.67	1.19	2.18	2.52
	10^3	**1.00**	1.45	1.15	1.29	2.37	1.71	1.89	2.13
	10^4	**1.00**	1.49	1.19	1.44	2.75	2.07	12.20	13.76
	10^5	**1.00**	1.29	1.09	2.16	3.61	2.84	NA	NA
	10^6	**1.00**	1.16	1.05	2.35	3.91	3.38	NA	NA
	Avg.	**1.00**	1.34	1.10	1.65	2.86	2.24	NA	NA
Total Avg.		**1.01**	1.43	1.14	2.74	3.52	2.76	NA	NA

Table XIV. Relative Performance for Exponential Distribution on AMD Athlon MP (Normalized with Respect to the Fastest Access Time, Where the Higher the Number, the Slower It Is)

Model	Queue Size	Henriksen with Twol	Splay Tree with Twol	Skew Heap with Twol	Henriksen	Splay Tree	Skew Heap	CQ	DCQ
Classic Hold	10^2	1.07	1.30	1.21	1.06	2.09	1.51	**1.00**	1.10
	10^3	1.08	1.37	1.37	1.38	2.83	1.83	**1.00**	1.08
	10^4	**1.00**	1.20	1.08	3.07	3.68	2.44	1.15	1.45
	10^5	**1.00**	1.11	1.06	4.33	5.88	4.76	1.38	1.51
	10^6	**1.00**	1.14	1.09	6.38	8.10	7.07	1.46	1.51
	Avg.	**1.03**	1.22	1.16	3.24	4.52	3.52	1.20	1.33
Up/Down	10^2	1.07	1.05	1.02	**1.00**	1.29	1.18	2.51	2.72
	10^3	1.01	1.13	**1.00**	1.15	1.75	1.51	1.85	2.02
	10^4	1.04	1.08	**1.00**	1.92	2.35	1.86	1.99	2.28
	10^5	1.04	1.00	**1.00**	2.47	3.08	2.47	1.59	1.71
	10^6	**1.00**	1.01	**1.00**	3.40	4.23	3.63	1.46	1.53
	Avg.	1.03	1.05	**1.00**	1.99	2.54	2.13	1.88	2.05
Total Avg.		**1.03**	1.14	1.08	2.62	3.53	2.83	1.54	1.69

Table XV. Relative Average Performance for All Distributions on AMD Athlon MP (Normalized with Respect to the Fastest Access Time, Where the Higher the Number, the Slower It Is)

Model	Queue Size	Henriksen with Twol	Splay Tree with Twol	Skew Heap with Twol	Henriksen	Splay Tree	Skew Heap	CQ	DCQ
Classic Hold	10^2	1.15	1.35	1.25	**1.00**	2.02	1.55	1.09	1.24
	10^3	1.04	1.30	1.14	1.24	2.27	1.71	**1.00**	1.30
	10^4	**1.00**	1.21	1.09	2.75	2.78	2.20	3.89	2.01
	10^5	**1.00**	1.09	1.07	6.29	4.16	4.31	4.35	2.35
	10^6	**1.00**	1.13	1.09	11.99	6.08	6.27	2.72	NA
	Avg.	**1.04**	1.22	1.13	4.65	3.46	3.21	2.61	NA
Up/Down	10^2	1.13	1.13	1.07	**1.00**	1.36	1.14	2.55	2.89
	10^3	1.02	1.10	**1.00**	1.04	1.59	1.38	1.81	2.01
	10^4	**1.00**	1.09	**1.00**	1.54	1.87	1.46	6.93	7.55
	10^5	1.04	1.02	**1.00**	2.17	2.64	2.17	NA	NA
	10^6	**1.00**	1.03	**1.00**	2.95	3.77	3.29	NA	NA
	Avg.	1.04	1.07	**1.01**	1.74	2.25	1.89	NA	NA
Total Avg.		**1.04**	1.15	1.07	3.20	2.85	2.55	NA	NA

ACKNOWLEDGMENTS

We thank SGI Singapore, especially Mr. David Tan, for allowing us to perform our benchmarks on the SGI servers. We have scaled up to 10 million events in the experiments done on the SGI Altix 3300, thus establishing further the validity that Twol-amalgamated priority queues offer good performance in large-scale application scenarios. Also, we extend our gratitude to Professor Catherine C. McGeoch for helping us to improve the quality of this article.

REFERENCES

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

BROWN, R. 1988. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Commun. ACM 24*, 12 (Dec.), 825–829.

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

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.

ERICKSON, K. B., LADNER, R. E., AND LAMARCA, A. 2000. Optimizing static calendar queues. *ACM Trans. Model. Comput. Simul. 10*, 3 (July) 179–214.

FALL, K. AND VARADHAN, K. 2002. The ns Manual. UCB/LBNL/VINT Network simulator v2. http://www.isi.edu/nsnam/ns/.

GOMES, F., FRANKS, S., UNGER, B., XIAO, Z., CLEARY, J., AND COVINGTON, A.. 1995. SimKit: A high performance logical process simulation class library in C++. In *Proceedings of the 1995 Winter Simulation Conference*, 706–713.

HAGAI, A. AND PATT-SHAMIR, B. 2001. Multiple priority, per flow, dual GCRA rate controller for ATM switches. In *IEEE Workshop on High Performance Switching and Routing*, 169–174.

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

HENRIKSEN, J. O. and Crain, R. C. 1996. GPSS/H Reference Manual, 4th ed. Wolverine Software Corporation, Annandale, VA.

HENRIKSEN, J. O. 1997. An introduction to SLX. In *Proceedings of the 1997 Winter Simulation Conference*, 559–566.

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

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

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

L'ECUYER, P. AND CHAMPOUX, Y. 2001. Estimating small cell-loss ratios in ATM switches via importance sampling. *ACM Trans. Model. Comput. Simul. 11*, 1 (Jan.), 76–105.

NARLIKAR, G. AND ZANE, F. 2001. Performance modeling for fast IP lookups. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1–12.

OH, S. AND AHN, J. 1998. Dynamic calendar queue. In *Proceedings of the 32nd Annual Simulation Symposium*, 20–25.

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

PRITSKER, A. AND PEGDEN, C. 1979. Introduction to Simulation and SLAM, 3rd ed. Wiley, New York.

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

RÖNNGREN, R. AND AYANI, R. 1997. A comparative study of parallel and sequential priority queue algorithms. *ACM Trans. Model. Comput. Simul. 7*, 2 (Apr.), 157–209.

SCHWETMAN, H. 1996. CSIM18 User's Guide. Mesquite Software, Inc, Austin, TX.
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.
STOICA, I., ZHANG, H., AND NG, T. S. E. 2000. A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services. *IEEE/ACM Trans. Networking* 8, 2 (Apr.), 185–199.
TARJAN, R. E. 1985. Amortized computational complexity. *SIAM J. Algebraic Discrete Meth. 6*, 2 (Apr.), 306–318.
THNG, I. L. J. AND GOH, R. S. M. 2004. Swan—Simulator without a name. http://swan.nus.edu.sg/.
YUGO, K. I. R., MOFFAT, A., AND NGAI, C. H. A. 2002. Enhanced word-based block-sorting text compression. In *Proceedings of the 25th Australasian Conference on Computer Science*, 129–137.

Received November 2003; revised September 2004, October 2004; accepted December 2004