A Heap Structure for the Simulation Event Set

Stavros D. Nikolopoulos
Department of Computer Science, University of Ioannina,
P.O. Box 1186, GR-45110 Ioannina, Greece
e-mail: stavros@cs.uoi.gr

Abstract—This paper introduces an efficient data structure, the block-heap structure, for maintaining future events in a general purpose discrete event simulation system. It combines the advantages of the static and dynamic heap representations providing a time / space efficient scheduling algorithm. To gain insight into the performance of the block-heap scheduling algorithm and allow comparisons with the static-heap and dynamic-heap algorithms, they are tested under a wide variety of conditions in an empirical / experimental way. For this purpose, a set of stochastic scheduling distributions are especially chosen to show the advantages and limitations of each algorithm. The processor time abstained with the block-heap algorithm is relatively insensitive in the scheduling distributions or the number of events notices in the event set. Experimental results and comparisons point to the superiority of the block-heap algorithm in respect of time performance and storage economy, and put strong emphasis upon the block-heap as an important scheduling algorithm for practical use. Finally, we show that the insertion of k new items in a block-heap of size N or the deletion of the k smallest items can be efficiently performed on the EREW PRAM model of computation with k processors, where $1 \leq k \leq N$.

Keywords: Discrete-event simulation, event scheduling algorithms, data structures, heaps, time/space behaviour, parallelism.

1. Introduction

In a discrete-event simulation system based on the *next-event time-advance* approach, an event set algorithm performs the following three main operations: (a) schedule future events, (b) cause the events to occur on the proper simulation time, and (c) update the clock variable which represents time. In such a system, an event is a collection of actions which are considered as executed in specific time. The data structure used to maintain future events in the system is usually called *event set* or *event list* and the time at which events are scheduled to occur is called *event time* or *schedule time*.

In the simulation environment, events are generally kept in objects known as *event notices*. An event notice is represented by a record with two fields, say t and a, where t is the scheduled time for its occurrence, and a is the activity which is scheduled in time t. When the time t of a new event becomes known, it is scheduled—that is, a event notice is created and it is inserted into the event set in a position ensuring that it occurs at the scheduled time t. On the other hand, when the *next-event* (i.e., the event with the minimum event time) occurs, it is removed from the event set, and the simulation clock is advanced to the time of the next-event. The responsibility for these operations is due to an algorithm which (i) scans the event set to determine the proper insertion position for the new event, (ii) removes the next-event from the event set, and (iii) advances the simulation clock to the time of the next-event. Such an algorithm which used to schedule events in a general purpose discrete event simulation system is known as an *event set algorithm* or *event scheduling algorithm*.

The above operations involved by an event scheduling algorithm, are the most frequent operations required by any discrete-event simulation system. It is obvious that, the scanning operation becomes quite expensive in "real-word" application of simulation where the number of schedule events in the event set becomes rather large. For "large" complex simulation systems which generate many events the execution time of an event scheduling algorithm is one of the most crucial factors of systems' performance. Although the factor memory is not so important for "small" systems, it becomes considerably more important as the number of event notices becomes large. It is therefore clear that the most important requires of an event scheduling algorithm are (i) speed of operation and (ii) storage economy.

There are two principal approaches to storing event notices in a computer system or, more precisely, there are two principal approaches of internal representations for structures and their special varieties: *contiguous-memory* or *static* representations and *linked* or *dynamic* representations. In the *contiguous-allocation* approach, the even notices are put into physically adjacent storage locations, one notice after another. In the *linked-allocation* approach, each event notice contains its usual attributes and, in addition, pointers giving the local relationship of the notices to other notices in the event set. The static and dynamic representations of lists of event notices have several advantages and disadvantages for simulation modelling. Some of them are the following:

- In a static representation, the event notices are inserted in the system and stored in contiguous memory location—that is, they are stored in an array whose size is fixed. Obviously, its size must be greater than or equal to the maximum number of event notices which should be inserted in the system. Thus, in a static representation a maximum size for the list must be predefined, whereas a dynamic representation do not have this limitation.

- Contiguous representations are more efficient in bytes required per event notice than linked representations, due to the absence of memory locations for pointers.

- Storage in contiguous memory is less attractive when the event notices have different lengths, because a specific event notice cannot be found in constant time using simple arithmetic.

- Linked representations are more flexible than contiguous representations, because only the pointers need to be adjusted in order to insert or delete event notices, and because the maximum size of the set is bounded only by the total memory available in the system.

- Linked representation provides a general framework that allows one to store and manipulate many event sets simultaneously with ease, whereby event notices in deferent event sets may be processed in different way. This generality is one of the reasons for the use of linked representations by all major simulation languages.

Many researchers have been published work present both analytical and empirical results concerning the time / space performance of many event scheduling algorithms. They use different data structures for the simulation event sets, e.g., linear lists, special kinds of trees, time-indexed lists, two-level structures, etc. Moreover, they use different techniques for the operations performed by the scheduling algorithms; for example, see [1, 7, 16, 24].

In this paper we introduce an efficient data structure, the *block-heap* structure, for the simulation event set which is based on the notion of complete binary trees. Specifically, it based in part on the static representation and in part on the dynamic representation of the heap structure. Thus, the block-heap structure combines the advantages of the contiguous-memory representation of heaps (*static heaps*) and linked representation of heaps (*dynamic heaps*), providing a time / space efficient scheduling algorithm.

To gain insight into the performance of the block-heap and allow comparisons with both static and dynamic heap scheduling algorithms, all the algorithms are coded and tested under a wide variety of conditions in an empirical / experimental way. The objective was to estimate the average complexity for each algorithm. For this purpose, we used a revised definition of complexity. That is, for a given configuration of event set and a given distribution providing the schedule time, we taken the processor time expended for the execution of the *R procedure* (or *R model*; see Section 8) as the complexity of the algorithm for that input. The R procedure combines the following operations:

(1) Determine the next-event—that is, the event notice with the minimum schedule time.
(2) With probability p, create a new event notice with schedule time T, and insert it into event set; Insertion or Scheduling operation.
(3) With probability 1-p, remove from the event set the event notice with the minimum schedule time; Removal operation.

Two main parameters affect the execution time of the above operations. They are (i) the schedule time T, and (ii) the size N of the event set (usually, we take $p = 0.5$). The parameter T, which is given by a stochastic distribution, determines how long an event will remain in the event set. Six stochastic distributions are especially chosen which are not only representative of typical simulation problems but also capable to show the advantages and limitations of each algorithm. All distributions have mean one and fall into three categories:

(A) Unimodal continuous distributions
(B) Bimodal continuous distributions
(C) Discrete distributions

The parameter N defines the notion of the *small* and *large* event sets. In other words, it determines the size of the event set—that is, the number of event notices in the set at any time. Tests were performed with values of N from 64 (small event set) to 4096 (large event set). This range is representative of actual simulations, and the behaviour for $N > 4096$ can be extrapolated from the results. The processor time obtained with the block-heap algorithm is relatively insensitive to variations in the scheduling distributions (parameter T) or the number of event notices in the event set (parameter N). The block-heap algorithm is faster than the dynamic-heap, but the static-heap. However, the contiguous representation of the heap structure may waste space since it uses a maximum amount of space which is independent of the number of event notices that are actually on the set at any time. On the other hand, the linked representation of the heap structure has a considerable space overhead. The results of this work shows that the block-heap algorithm combines time performance, storage economy and simplicity of coding. As a consequence the block-heap structure is well suited to incorporation in general purpose simulation languages; see [5, 6, 14, 18].

The paper is organized as follows: Section 2 presents two data structures, the static-heap and dynamic-heap, which are both suitable for the simulation event set. Section 3 introduces the proposed block-heap structure. Section 4 presents all the event scheduling algorithms tested—that is, the static-heap, the dynamic-heap and the block-heap algorithms. Section 5 describes the R model and gives the conditions under which the algorithms were tested. Section 6 reports the experimental results and explain them, while Section 7 provides a comparison analysis of all the algorithms. Section 8 shows that the block-heap is a suitable structure for parallel insertion of k new elements or deletion of the k smallest ones on the EREW PRAM model with k processors. Section 9 concludes the paper.

2. The Heap Structure and Heap Property

A *left-complete binary tree* is a binary tree with external nodes on at most two adjacent levels such that the external nodes on the lowest level are leftmost. It is obvious that, for each nonnegative integer N, there is only one binary tree of size N that is left complete. Now a *heap* is a left-complete binary tree, each of whose nodes includes an element of information called the value of the node, and which has the property that the value of each internal node is smaller (or greater) than or equal to the values of its children. This is called the *heap property* [3, 8, 12, 15]. We shall use the term *heap structure* to describe a left-complete binary tree that satisfies the heap property.

We next outline both the dynamic and static representations of the heap structure. We shall refer to these two representations as *dynamic-heap* and *static-heap*, respectively.

2.1 The Dynamic-Heap Structure

On a computer, any left-complete binary tree may be represented using nodes with three fields, say *info*, *lp* and *rp*—that is, in such a tree, each node has an information field *info*, and two pointers *lp* and *rp*, one to its left child and one to the right child, either of which can be nil. Notice that, by definition, a left-complete binary tree can have 0, 1, or 2 children and if *lp* is nil then *rp* is also nil.

This representation can be used by any tree; it has the advantage that all the nodes can be represented using the same record structure, no matter how many children they have. However many operations are inefficient using this minimal representation; for example, it is not obvious how to find the parent of a given node efficiently. A suitable representation for our particular application can be designed by adding a supplementary pointer *bp* to parent of each node. As we shall see, using this pointer, we speed up the insertion and deletion operations at the price of an increase in the storage needed.

2.2 The Static-Heap Structure

Due to their properties, there are several ways other than the usual linked or dynamic structures to represent left-complete binary trees, among which the contiguous-memory or static representation. We can represent heaps (left-complete binary trees having the heap property) implicitly in an array without the use of pointers by simply putting the root at position 1 and the direct descendants of the node i at positions $2i$ and $2i+1$; see [15]. Specifically, a heap of N nodes is represented by an array A[1..N], where the heap property $A[i] \geq A[i \text{ } div \text{ } 2]$ holds for any i, $2 \leq i \leq N$; *div* denoting integer division.

3. The Block-Heap Structure

A major problem with the static representation of the heap structure is that it restricts simulation systems to handle a predefined number of event notices. That is, a maximum size decision must be made in advance by the system. One natural way to avoid this restriction is to use pointer representation. Although, in principle, we can implement a heap as an explicit tree structure with parent and children pointers (dynamic-heap), it is less efficient than the static-heap due to the extra space used by the pointers and extra time taken by the memory manager for the allocation / deallocation process during a insertion / deletion operation.

We propose here a heap structure that overcame the static and dynamic drawbacks. It combines logarithmic behaviour, space efficiency and simplicity, and, thus, it can be efficiently used for the simulation event set. This structure is called *block* heap or *B*-heap.

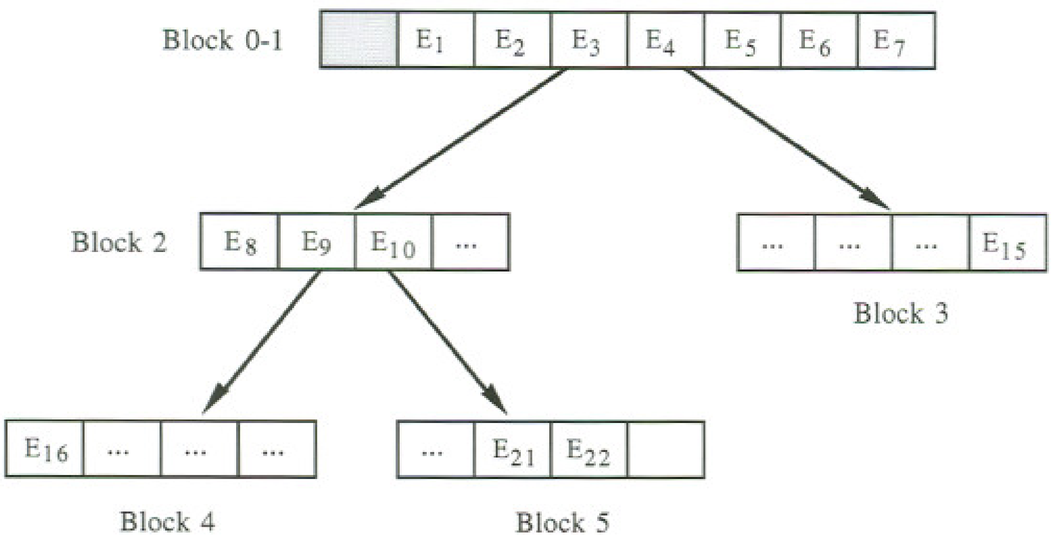


Fig. 1: A Block-heap structure; it consists of six blocks of length 4 and maintains twenty two event notices. The root of the heap consists of a block of length 8.

The block-heap is based in part on the static representation of the heap and in part on the dynamic one. The structure is simply a left-complete binary tree having nodes with $b+2$ fields (except the root of the tree), say E_1, E_2, ..., E_b, *lp* and *rp*—that is, each node of the tree has b information fields, and two pointers *lp* and *rp*, one to its left child and one to the right child. The root node has $2b$-1 information fields and two pointers; its first information field remains always empty. Of course, there is a pointer which gives reference to the root of the tree (see Figure 1). The nodes of a such structure are called *blocks*.

In this structure there are as many pointers as blocks. Each block, except the first (root) and possibly the last, maintains b elements at positions 0, 1, 2, ..., b-1. The kth block is denoted by B[k], while the ith position of the kth block is denoted by B[k, i], $0 \leq i \leq b\text{-}1$ and $k \geq 0$. Sometimes, we shall say "the block k" instead of "the kth block". The ith level of a block heap contains 2^i blocks—that is, the blocks B[2^i], B[2^i+1], ..., B[$2^{i+1}-1$], where $1 \leq i \leq h\text{-}1$ and $h = \log(N/b)$ is the height of the block heap. The level 0 contains a block of size $2b$ or, equivalently, the blocks B[0] and B[1], while the hth level contains m blocks—that is, the blocks B[2^h], B[2^h+1], ..., B[$2^h+(m-1)$], where $1 \leq m \leq 2^{h+1}$. The block heap of Figure 1 consists of six blocks of length $b = 4$; its root consists of a block of length $2b = 8$. Notice that the position B[0, 0] remains empty. The reason is that it makes the operations of finding the parent or a child of a node very easy (see the formulae which are presented in the next paragraph).

Let T be an ordinary heap of length N, and let A[1], A[2], ..., A[N] be the static representation of T. We can store the elements of the heap T in a block heap having blocks of length b as follows:

1. The root r = A[1] of the heap T is stored in the 1st position of the block 0—that is, A[1] is stored in B[0, 1], while the elements in A[2], ..., A[b-1], A[b], ..., A[$2b$-1] are stored in B[0, 2], ..., B[0, b-1], A[1, b], ..., A[1, $2b$-1], respectively. The element in A[$2b$] is stored in B[2, 0].

2. If a node u ($u \neq r$) is stored in the ith position of the block $k \geq 2$, then
   - the parent $p(u)$ of u is stored at position (i *div* 2) + b / 2 (k *mod* 2) of the block k *div* 2.

3. If a node u ($u \neq r$) is stored in the ith position of the block $k \geq 2$, then its children (if they exist) are stored as follows:
   - the left child of u is stored at position $2(i \bmod (b/2))$ of the block $2k + (i \operatorname{div} (b/2))$.
   - the right child of u is stored at position $2(i \bmod (b/2)) + 1$ of the block $2k + (i \operatorname{div} (b/2))$.
4. In any case there are at most b-1 empty block positions.

The above formulas make it easy to implement algorithms that traverse a block heap moving from a node to its parent or a child in various ways.

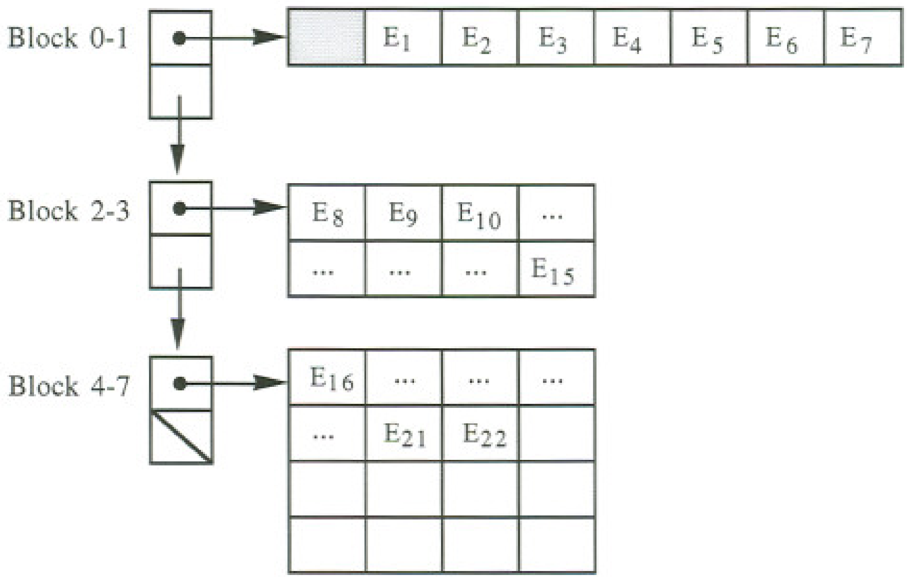


Fig. 2: An *L*-block heap structure; it consists of three blocks of lengths 8, 8 and 16, respectively. It maintains twenty one event notices. The root of the heap consists of two blocks.

Let *B*-heap be a block heap of size N having blocks of length b. By construction, this structure contains as many pointers as blocks. We can reduce the number of pointers by simply grouping all the blocks of the same level and placing them in a "chain". If we do this, we take the structure which consists of the following components: (1) same arrays of lengths $k\,b$, where $k = 2^i$ and $i \geq 1$; these arrays are called *blocks*; (2) a linked list of pointers that indicate the blocks; it is of length $h = \log(N/b)$; (3) two pointers that give reference to the head and tail of the linked list (see Figure 2). We call this structure *L*-block heap or *LB*-heap.

Again, we denote by B[i] the block which is pointed by the pointer of the ith node of the linked list of an *L*-block heap, $i \geq 1$. It is easy to see that the length of its linked list equals the height of the block heap; of course, this is true in the case where we have stored N elements in both heaps. Moreover, it is also easy to see that if a node u ($u \neq r$) is stored in the block B[i], then the parent (resp. children) of u is stored in the block B[i-1] (resp. B[i+1]).

Notice that we can easily modify the formulas for finding the direct descendants or the parent of a node in the block-heap structure, so that they can be applied to the *L*-block heap structure.

4. Event Set Algorithms

The responsibility for deletion the notice with the minimum event time (next-notice) from the event set and insertion a new notice into event set is due to an event scheduling algorithm. Such an algorithm simply scans the event set to find (and remove) the next-notice and determine the proper insertion position for the new event notice. In this section we present the event scheduling algorithms that perform these operation on the data structures described above. Hereafter, the event notice with the minimum event time is referred to as *current notice*.

4.1 The Dynamic-Heap Algorithm

We have seen that a way to represent a heap—that is, a left-complete binary tree having the heap property, is that of using nodes with four fields, say *info*, *lp*, *rp* and *bp*, where the field *info* of node u contains an event notice and the tree fields *lp*, *rp* and *bp* contain pointers to its left child, right child and parent of u, respectively (dynamic representation).

Under the heap property, the current notice is always placed in the root of the heap and, therefore, it can be found in constant time. For the deletion operation, the algorithm copies the event notice of the rightmost nonvacant leaf of the last level into the root and decreases the size of the heap by 1—that is, the right most leaf node is deleted from the heap. The event notice in the root is now moving back towards the leaves via the eldest son at each level until the heap property is re-established. The insertion operation introduces a new external node; that is, a leaf. Specifically, the algorithm takes the current notice from the root of the heap and increases its event time—say t, by T (R parameter). Then, create a new event notice with event time $t+T$ and place it in the leftmost vacant leaf of the heap. From there, the algorithm moves the new notice towards the root of the heap (using the *bp* pointer) until it blocked by an event notice with an earlier or the same event time.

It is very easy to estimate the worst case behaviour of a heap. Obviously, the operation of finding the current notice takes time $O(1)$. Moreover, it is well known that the height of a heap containing N nodes is $\lfloor \log N \rfloor$ and, therefore, the operations of deleting the current notice or inserting a new one takes time $O(\log N)$ in the worst case.

2.2 The Static-Heap Algorithm

Although, in principle, we can implement the left-complete binary tree having the heap property as an explicit tree structure with parent and children pointers, we can implement it more efficiently without theses pointers (static representation). Thus, using the static representation for the heap structure, the current notices are stored in an array A of length Nmax. Again, under the heap property the current notice is always placed in A[1].

We use the same procedure for the insertion and deletion operation as that we use in the dynamic-heap. The root of the heap is now the element A[1], while its right most node is the element A[N], where N is the size of the heap. Notice that, the insertion operation increases the size of the array A by 1 and places the new event notice (with event time $t+T$) in A[N+1]. (Recall that N is the number of event notices in the array, where $N <$ Nmax.) The deletion operation copies the notice of A[N+1] in A[1] and decreases the size of the array A by 1. In both cases, the event notice in A[1] or A[N+1] are moving back towards the leaves or the root of the heap via the eldest son at each level until the heap property is re-established.

It is obvious that the operations of finding the current notice still takes time $O(1)$ and deleting the current notice or inserting a new one still takes time $O(\log N)$ in the worst case.

4.3 The Block-Heap Algorithm

We have introduced the block-heap as an efficient structure for the simulation event set and presented it in Section 3. We now describe the insertion and deletion operations performed by the block-heap event scheduling algorithm. We shall concentrate on the block heap structure (we can easily extent these operations on the *L*-block heap).

In the block-heap structure the current notice is always placed in the root of the heap—that is, B[0, 1], since it satisfies the heap property. Let as consider the insertion and deletion operations. Suppose that there are *N* event notices in the block-heap and *b* the length of its blocks. Insertion take place by incrementing *N* and placing the new event notice in B[(*N*+1) *div b*, (*N*+1) *mod b*] initially. From there it moves via its ancestor towards the root B[0, 1], until blocked by a notice with an earlier or the same event time. For this movement the father pointers are used (see dynamic heap), since the ancestor of a node is located in a different block. Removing the current notice from B[0, 1], the position B[0, 1] is refilled from B[*N div b, N mod b*] and *N* is decreased. The event notice in B[0, 1] now moves back towards the leaves via the earliest son at each level, until blocked by one with a late event time. For this movement the children pointers are used and the appropriate formula to determine the position which hold the earliest son.

As far as the complexity of the algorithm is concerned, it is easy to see that the worst-case complexity of both insertion and deletion operations are $O(\log N)$, where N is the size of the linked-heap.

5. Testing the Algorithms

Closely coupled to the search for efficient scheduling algorithms is the problem of testing. We are interested in testing the proposed linked-heap scheduling algorithm, as well as the linked-list and heap algorithms, under a wide variety of conditions in an empirical / experimental way, in order to estimate their average time complexity. In this section we present the conditions under which all the algorithms were tested.

5.1 The R Model

The most frequent operations required by any discrete event simulation system are *deletion* of the current notice from the event set and *insertion* of a new notice into the event set. Most of the research work performed to date uses the HOLD procedure which combines these two basic operations, to estimate the average time complexity of the algorithms [7, 8, 19, 24]. Using the HOLD operation, it is clear that the size of the event set does not change (a deletion is followed by an insertion). This is not very realistic, since in real simulations the size of the event set varies dynamically.

In this work, we use a more realistic model—that is, the *R* model, to estimate the average time complexity of our algorithms. The R procedure first determines the event notice with the minimum event time *t* and then either removes it from the event set with probability *p* (deletion operation), either increases its event time *t* by *T*, where *T* is the random variant distributed according to some distributions, and reinsert it with the new value of event time *t*+*T* into the event set with probability 1-*p* (insertion operation). The difference between the new and old values of the event time is passed to the R procedure as the parameter *T*. Throughout the paper the parameter *T* is referred to as *R parameter* and the processor time (CPU time) for the execution of the R procedure is referred to as *R time*.

Table I: The Six Distributions

(A) Unimodal continuous
- (1) Negative exponential (mean 1).
- (2) Uniform distribution over the interval [0, 2].
- (3) Uniform distribution over the interval [0.9, 1.1].

(B) Bimodal continuous
- (4) 0.9 probability - uniform over the interval [0, S],
  0.1 probability - uniform over the interval [100S, 101S],
  where S is chosen to give the mixed distribution an average of unity.

(C) Discrete
- (5) T is constant with value of unity.
- (6) T is assigned the values 0, 1 or 2 with equal probabilities.

5.2 Test Conditions

One parameter which affects the execution time of the R operation is the size N of the event set. We are interested in testing the performance of each algorithm using *small* and *large* event sets. Tests were performed for N = 64, 128, 256, 512, 1024, 2048 and 4096. This range of sizes is representative for actual simulations and the behaviour of the algorithms for $N > 4096$ can be extrapolated for the results.

A crucial step in designing the tests lies in the selection of stochastic distribution which provides the event time T. The stochastic distribution used to determine how long an event notice will remain in the event set. Six distributions have been chosen as in [7, 8]. They differ widely in their characteristics and provide an efficient test for the good and bad points of the algorithms. All chosen distributions have mean 1 with different variances and fall into three categories; see Table I.

As the measure for estimating the average time complexity of the algorithms we have taken the processor time (CPU time) for the execution of the R operations. In particular, for a given configuration of event set and a given R time, say T, we have taken the processor time expended for the execution of R(T, p) as the complexity of the algorithm for that input. Since the R operations is affected by the size of the event set and the stochastic distribution, we have executed it with various combinations of event set sizes and distributions.

The algorithms were programmed in C and the test were performed on a SUN Enterprise 450 running under Solaris version 2.6. The number of R operations carried out was constant for all N and equal to 8000—that is, $n_i + n_d = 8000$, where n_i and n_d are the numbers of insertions and deletions, respectively.

The sequence of operations in each test was as follows:

(i) Generate and insert N event notices into the event set. Initialize each event notice with event time generated from the chosen distribution F.

(ii) Before starting the time count, execute k times the R procedure with the same distribution F and $p = 0.5$, where $k = 2N$.

(iii) Execute 8000 R operations and count the total processor time (CPU time).

Operation (i) initializes the system. Operation (ii) is executed to permit the system to reach steady state. In analytical work, it is assumed that k is infinity. In practice, k is some small multiple of the size of the event set. In our case $k = 2N$. Operation (iii) yields a measure of the complexity of the tested algorithms.

The time required to generate the random numbers, perform the loops and call the R procedure was subtracted from the running time so that the CPU times are comparable from one distribution to another. The random numbers generator used in all distributions in based on the Lehmer's congruential method [9] and is reported and tested by Downham and Roberts [4]. The generator is coded in such a way as not to give overflow on 32-bit computer.

6. Presentation of the Results

Each of the algorithms was coded and run to collect evidence of performance under realistic conditions. The experimental results for each algorithm are represented in figures in the form of graphs of total CPU time against the size N of the event set. For all algorithms a logarithmic scale has been used in their graphs. The graphical results showing the time performance of the *static-heap, dynamic-heap* and *block-heap* algorithms are presented in Figure 3. The real processor time (CPU time) taken by the algorithms are showed in Tables III, IV and V; see Appendix.

In general, we can say that the heap algorithms provide a very good time performance. In all cases we observe the expected $O(\log N)$ time complexity (logarithmic behaviour) of these algorithms. An important point it should be stressed is that the variation of CPU times with N, given by the slope of each line, increases as the variance of the scheduling distribution decreases. It was expected because the heap algorithms scan forward the event notices for the removal operation—that is, from small to large event time. We also observe that the time performance of these algorithms is almost the same with all the distributions. Specifically, it is best with both *Discr*(0, 1, 2) and *Discr*(1) distributions (FIFO distributions) and worst with *Exp*(1) and *Bimodal*.

As far as the behaviour of the block-heap algorithm is concerned, we observe that it performs very well with all the sizes and distributions. Specifically, it gave the expected $O(\log N)$ behaviour in all cases. As with the other heap algorithms, the variation of CPU times with N given by the slope of each line, increases as the variance of the scheduling distribution decreases. It was an expected similarity since all the algorithms use the same technique for the insertion and deletion operations.

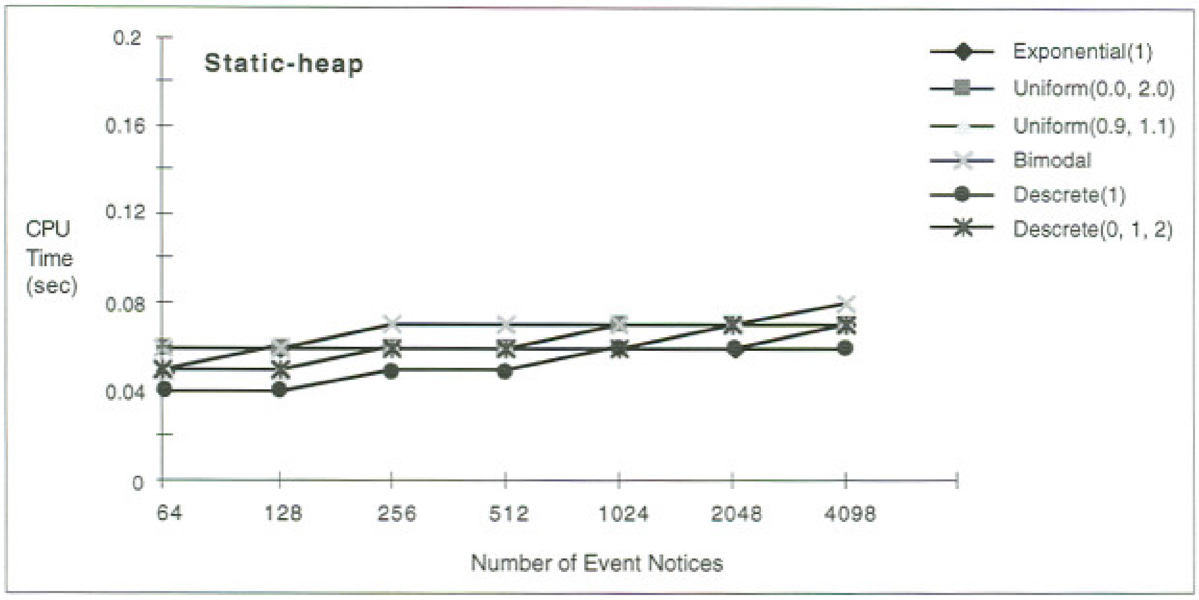
It should be pointed out that the static-heap algorithm gives a slight improvement over the block-heap algorithm. Its better behaviour is caused of the fact that it uses a static representation while the block-heap uses a combination of static and dynamic representations.

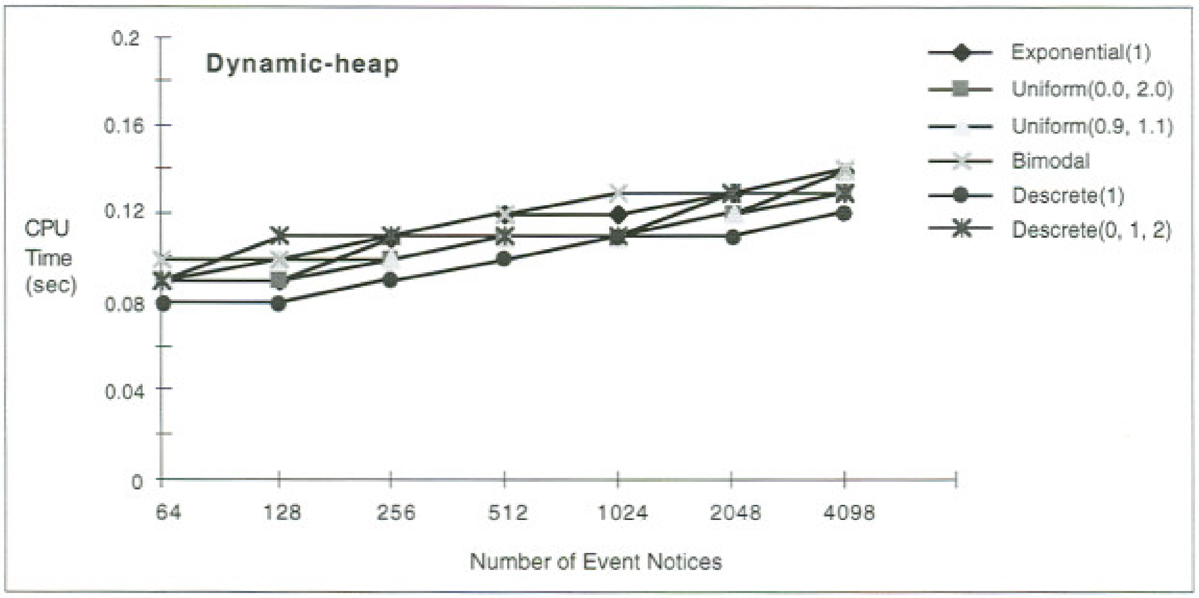
The block-heap algorithm was implemented having block size $b = 4$. Tests with others values of theses parameters is a topic for future work.

Although the graphical presentations of the results show the time performance of the algorithms tested clearly, we include here (see Appendix) the real processor time (CPU time in *sec*) taken by the algorithms, for more clarity and accuracy. Tables III, IV and V correspond to the empirical mean complexity for the dynamic-heap, static-heap and block-heap algorithms, respectively. The rows correspond to the value of N and the columns to the choice of distribution delays.

7. Static-Heap and Dynamic-Heap vs Block-Heap

In this section we compere the static-heap and dynamic-heap structures with the block-heap structure with respect to time performance and space economy. The comparison results show that the block-heap is, in general, an efficient structures for the simulation event set.





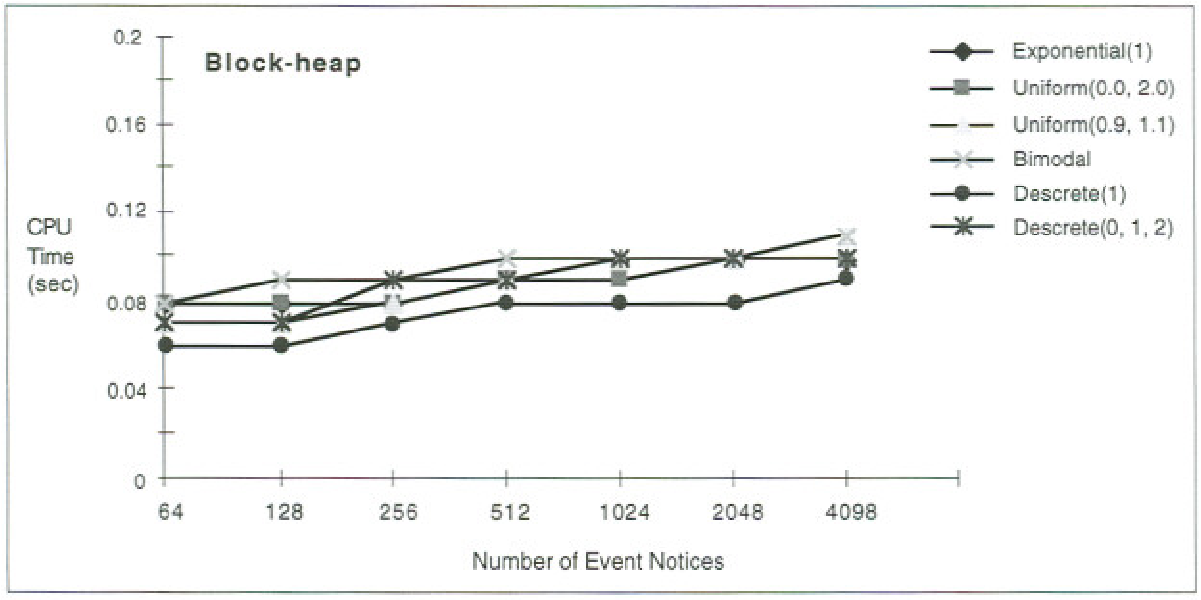


Fig. 3: The Static-heap, Dynamic-heap and Block-heap algorithms; test / experimental results.

Specifically, the array representation (static-heap) uses array indexing, which is often faster than pointer representation (dynamic-heap). Moreover, it avoids the space overhead of pointers. But all static representations have a predefined maximum size, whereas dynamic representations do not have this limitation. Hence, the static-heap is preferred whenever we know the maximum possible size of the list of events in a simulation system; in all other situations, the dynamic-heap or block-heap are preferable. In general, the unknown amount of events that is stored and manipulated by a discrete event simulation system dictates clearly the use of a dynamic approach (not dynamic representation) for storing events in such a system.

The dynamic-heap structure requires a large amount of memory locations. Furthermore, it has a time overhead associated with the pointer manipulation process. On the other hand, the static-heap provides excellent time performance for both insertion and deletion operations, but it suffers for inability to maintain any amount of events. That is, the number of events in the system is bounded by $N\text{max}$, where $N\text{max}$ is the predefined length of the array used to represent the heap. Additionally, extra space is reserved in the case where the number of events in the system is less that $N\text{max}$.

We can overcame the above drawbacks adopting the block-heap structure which incorporate both static and dynamic memory-allocation approaches. It provides very good (logarithmic) time performance, size flexibility and space economy. Moreover, the experimental results showed that its performance is relatively insensitive to variations in the scheduling distributions or the number of event notices in the event set. We realize that the static-heap performs better that the block-heap, but the deference is not noticeable.

Table II shows the memory needed by each algorithm. For comparison reasons, we also include in this table the memory needed by the L-block heap. In this tabulation, m and p indicate the memory needed to store a data (event notice) and a pointer, respectively. N is the number of events in the set and $N\text{max}$ is the predefined maximum length of the array used to represent a heap (static representation).

Table II: Memory needed by each Algorithm

Structure	Data space	Pointer space
Static-heap	$m\,N\text{max}$	0
Dynamic-heap	$m\,N$	$3\,p\,N$
Block-heap	$m\,(N + k),\ 0 \le k < b$	$3\,p\,N\,/\,b$
L-block-heap	$m\,(N + k),\ 0 \le k < N$	$3\,p\lfloor \log N \rfloor$

Looking at table II, we observe that the block-heap and dynamic-heap structures need almost the same memory space. In the worst-case the block-heap uses only b-1 extra memory locations than the dynamic-heap. Moreover, we observe the waste of data space of the static-heap in the case where the number N of events in this structure is much smaller than $N\text{max}$. As far as the pointer space is concerned, it is clear that the static-heap is out of competition since it uses no pointers. Comparing the remainder structures, we observe the superiority of the block-heap in respect of storage economy.

The comparisons made in this section between static, dynamic and block heaps give evidence of using the block-heap structure for the simulation event set.

8. Block-Heap and Parallelism

In this section we show that the block-heap structure can also be efficiently used in a parallel process environment. Specifically, we show that the block-heap allows efficient simultaneous insertions of k new items (i.e., event notices), or removal of the k items associated with the k smallest values (i.e., event time), where $k \leq N$ and N is the size of the block-heap. As a model of parallel computation we shall use the well known EREW PRAM; see [10, 23].

Many researchers have been devoted to the study of priority queue operations in a parallel environment [11, 20, 22]. Recently, Pinitti and Pucci [20] proposed a heap structure for the parallel insertion of b new items or deletions of the b smallest ones on the CREW PRAM model with b processors. If their structure contains N items, then the insertion can be performed in parallel time $O(h + \log N)$, while the deletion can be performed in time $O(h + \log\log N)$, where $h = \log(N / b)$. Notice that, the parameter b is strongly depended on the proposed heap structure.

Using a block-heap of size N with block length b and a EREW PRAM model of computation, we next show that k new items can be inserted in time $O(h)$ using k processors and the k smallest items can be deleted in time $O(h + \log N \log\log N)$ using $max\{k, N/\log N\}$ processors, where $h = \log(N/b)$. We can insert or delete as many items as the size of the block-heap—that is, $1 \leq k \leq N$.

8.1 Parallel Insertion

Let as now insert k items into a block-heap of size N having blocks of length b. Without loss of generality we assume that $k = i\, b$, where $i = 1, 2, ..., N / b$. We first place the k new items (event notices), in the i consecutive leftmost vacant leafs of the block-heap—that is, the blocks N/b, $(N/b)+1$, ..., $(N/b)+i$. Here, for simplicity, suppose that these blocks belong to the same level. Using now k processors, one for each new event notice, they move the new event notices towards the root of the heap until each of them blocked by an event notice with an earlier or the same event time. It is obvious that, each processor re-establishes the heap property. These operations are described by the following algorithmic scheme:

```
1. For all the new event notices at level $h = \log(N / b)$ do the following:
   1.1 Compere the even-indexed new event notices at level $h$ with their parents at level $h$-1
       and make the appropriate changes.
   1.2 Compere the odd-indexed new event notices at level $h$ with their parents at level $h$-1
       and make the appropriate changes.
   end;

2. For $i = \log(N / b)$-1, ..., 2, 1 do the following:
       Compere all the new event notices at level $i$ with their parents at level $i$-1 and make the
       appropriate changes.
   end;
```

After the execution of the step 1 of the above algorithmic scheme, same of the new notices are place at level h and same at level h-1, and no two of them are placed in consecutive locations in the same block. Moreover, based on the structure of the block heap, it is easy to see that during the execution of the step 2 no pair of new event notices satisfies the sibling-relationship—that is, no two event notices have the same parent. Thus, we conclude that the heap property is re-established after the execution of the above operations. Moreover, we can easily observe that no concurrent-read capability is required by the insertion; however, no concurrent-write capability is needed. So, the insertion operation can be executed on the less powerful PRAM model of computation—that is, the EREW PRAM.

The height of a block-heap containing N nodes is $h = \log(N / b)$ and, therefore, the operation of inserting k new items into a block-heap takes time $O(h)$ when k processors are available on the EREW PRAM model of computation, $1 \leq k \leq N$.

8.2 Parallel Deletion

We now show the way we can delete the k smallest event notices from a block-heap of size N having blocks of length b. Again, we assume that $k = i\,b$, where $i = 1, 2, ..., N / b$.

It is well-known that the block-heap (or, in general, the heap structure) do not support key ordering automatically. Under the heap property, only the event notice with the minimum event time is placed in known position—that is, position B[0, 1] of the root of the block-heap. Thus, the other k-1 smallest event notices should be located anywhere in the heap.

Suppose that the positions of the k smallest event notices are available, and let $p_1, p_2, ..., p_k$ be their positions (later, we show how to find these positions). In doing the deletion operation, we first copy the k event notices of the i consecutive rightmost nonvacant leafs of the block-heap—say, $a_1, a_2, ..., a_k$, into $p_1, p_2, ..., p_k$ positions and decreases the size of the heap by k—that is, the i rightmost blocks are deleted from the block-heap. Now, in order to re-established the heap property, some of the a's event notices are moving back towards the leaves via their eldest sons at each level, and some others are moving towards the root of the block-heap until blocked by event notices with earlier or the same event times. These movements can be easily done sing k processors, and are formally described by the following algorithmic scheme:

```
1. For log(N / b) iterations do the following:
   1.1  for i = 1, 3, ..., ⌈log(N/b)⌉-1 do:
                 Compere all the even-indexed a's notices
                 at level i with their parents at level i-1 and make the appropriate changes;
                 Do the same operation for all the odd-indexed a's notices at level i;
   1.2  for i = 2, 4, ..., ⌈log(N/b)⌉-1 do:
                 Compere all the even-indexed a's notices
                 at level i with their parents at level i-1 and make the appropriate changes;
                 Do the same operation for all the odd-indexed a's notices at level i;
   end;
```

Base on the well-known odd-even transposition sort algorithm [10, 23], we can easily show that after $\log(N/b)$ repetitions of the steps 1.1 and 1.2, each of the a's event notices satisfies the heap property. Thus, the above algorithmic scheme re-establishes the heap property in the block-heap after the deletion of the k smallest event notices. Moreover, we observe that during the copy process of the a's event notices into the p's positions and the execution of the steps 1.1 and 1.2 no concurrent-read or concurrent-write capability is needed. Therefore, the process of copying the a's event notices into p's positions and the steps 1.1 and 1.2 can be executed on the EREW PRAM model of computation; in total, they require time $O(h)$ and k processors, where $h = \log(N / b)$.

The previous discussion was based on the assumption that the positions of the k smallest event notices are available. We now show how to find these positions. It is obvious that we have to solve a selection problem—that is, the problem of determining the kth smallest elements of a sequence of N elements listed in arbitrary order. In the parallel environment, the selection problem can be solved in time $O(\log N \log\log N)$ with $N/\log N$ processors on the EREW PRAM model.

Thus, we conclude that the operation of deleting the k smallest event notices from a block-heap of size N can be performed in time $O(h + \log N \log\log N)$ with $max\{k, N/\log N\}$ processors on the EREW PRAM model of computation, $1 \leq k \leq N$.

9. Concluding Remarks

The block-heap structure presented in this paper could usefully replace the heap (static or dynamic representation) structures for the simulation event set in a general purpose discrete event simulation system. The processor time obtained with the block-heap algorithm is relatively insensitive to variations in the scheduling distributions or the number of event notices in the event set, and points to its superiority over the dynamic-heap. Also, the memory needed by this structure points to its superiority over the static-heap; using the block-heap we avoid to predefine the size of the event set. In general, it provides time efficiency, size flexibility and space economy. Moreover, we showed that the block-heap structure can also be efficiently used in a parallel process environment.

It is worth noting that, many articles have reported major improvements in execution time by implementing event scheduling algorithms other than the heap. [2, 6-9, 13, 17-19, 21]. Some of them provide a constant insertion time independent of the number of event notices in the event set. Best performance is achieved with the indexed-list and the two-level scheduling algorithms [7, 8, 24]. However, theses algorithms are unsuitable for a general purpose discrete event simulation system for several seasons (e.g., space consuming, setting parameters, etc.). For example, before the indexed-list can be used in a system, an adaptive mechanism must be devised to set some parameters (interval parameters) according to the operating conditions; the block-heap algorithm does not have such limitations. Of course, we should mention the commonly used linked-list algorithm. It provides simplicity, but both analytical and experimental results confirm the linear behaviour of these structure. The logarithmic behaviour of the block-heap structure points to its superiority over the linked-list in respect of time performance.

In closing, we point out that the results of this work prompts us to suggest the block-heap as an efficient data structure for the simulation event set in a general purpose discrete event simulation system.

References

[1] M. Andreou and S.D. Nikolopoulos, "An Efficient Data Structure for Maintaining Future Events in a Discrete-Event Simulation System", *2nd IMACS International Conference on Circuits, Systems and Computers (IMACS-CSC'98)*, Piraeus, 1998.

[2] R. Brown, "A Fast $O(1)$ Priority Queue Implementation for the Simulation Event Set Problem", *Comm. of ACM* 31 (12), 1220-1227, 1988.

[3] E-E. Doberkat, "Inserting a New Element into a Heap", *BIT* 21, 255-269, 1981.

[4] D.Y. Downham and F.D.K. Roberts, "Multiplicative Congruential Pseudo-random Number Generators", *The Computer Journal* 19(1), 74-77, 1975.

[5] G.S. Fishman, *Concepts and Methods in Discrete Event Digital Simulation*, John Wiley & Sons, 1973.

[6] W.R. Franta, *The Process View of Simulation*, Elsevier North-Holland, New York, 1977.

[7] W.R. Franta, and K. Maly, "An Efficient Data Structure for the Simulation Event Set", *Comm. of ACM* 20(8), 569-602, 1977.

[8] W.R. Franta, and K. Maly, "A Comparison of Heap and the TL Structure for the Simulation Event Set", *Comm. of ACM* 21(10), 873-875, 1975.

[9] A.T. Fuller, "The Period of Pseudo-random Numbers Generated by Lehmer's Congruential Method", *The Computer Journal* 19(2), 173-177, 1978.

[10] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[11] D.W. Jones, "Concurrent Operations on Priority Queues", *Comm. of ACM* 32(1), 132-137, 1989.

[12] G.H. Gonnet, "Heaps Applied to Event Driven Mechanisms", *Comm. of ACM* 19(7), 417-418, 1976.

[13] A. Jonassen and O-J. Dahl, "Analysis of an Algorithm for Priority Queue Administration", *BIT* 15, 409-422, 1975.

[14] A.M. Law and W.D. Kelton, *Simulation Modeling & Analysis*, McGraw-Hill, Inc., 1991.

[15] H.R. Lewis and L. Denenberg, *Data Structures & Algorithms*, Harpel Collins Publishers, New York, 1991.

[16] B.D. Lubachevsky, "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks", *Comm. of ACM* 32(1), 111-123, 1989.

[17] W.M. McCormack and R.G. Sargent, "Analysis of Future Event Set Algorithms for Discrete Event Simulation", *Comm. of ACM* 24(12), 801-812, 1981.

[18] I. Mitrani, *Simulation Techniques for Discrete Event Systems*, Cambridge University Press, 1982.

[19] S.D. Nikolopoulos and R. MacLeod, "An Experimental Analysis of Event Set Algorithms for Discrete Event Simulation", *Microprocessing and Microprogramming* 36(2), 71-81, 1993.

[20] M.C. Pinotti and G. Pucci, "Parallel Priority Queues", *Inform. Proc. Lett.* 40(1), 33-40, 1991.

[21] C.M. Reeves, "Complexity Analyses of Event Set Algorithms", *The Computer Journal* 27 (1), 72-79, 1984.

[22] V.N. Rao and V. Kumar, "Concurrent Access of Priority Queues", *IEEE Trans. Comput.* 37(1), 1657-1665, 1988.

[23] J. Reif (editor), *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1993.

[24] J.G. Vaucher and P.A. Duval, "A Comparison of Simulation Event List Algorithms", *Comm. of ACM* 18(4), 223-230, 1975.

Appendix

For more clarity and accuracy we include here the real processor time (CPU time) taken by the algorithms. The time taken by the Dynamic-heap, Static-heap and Block-heap algorithms are showed in Tables III, IV and V, respectively. The rows correspond to the value of *N* and the columns to the choice of distribution delays.

Table III: The Static-heap Algorithm; CPU Time

Size \ Distrib	EXP(1)	U(0.0, 2.0)	U(0.9, 1.1)	BIMODAL	DISCR(1)	DISCR(0, 1, 2)
N = 64	0.05	0.06	0.05	0.06	0.04	0.05
N = 256	0.06	0.06	0.06	0.07	0.05	0.06
N = 1024	0.06	0.07	0.06	0.07	0.06	0.06
N = 4096	0.07	0.07	0.07	0.08	0.06	0.07

Table IV: The Dynamic-heap Algorithm; CPU Time

Size \ Distrib	EXP(1)	U(0.0, 2.0)	U(0.9, 1.1)	BIMODAL	DISCR(1)	DISCR(0, 1, 2)
N = 64	0.09	0.09	0.09	0.10	0.08	0.09
N = 256	0.11	0.10	0.10	0.11	0.09	0.11
N = 1024	0.12	0.11	0.11	0.13	0.11	0.11
N = 4096	0.14	0.13	0.14	0.14	0.12	0.13

Table V: The Block-heap Algorithm; CPU Time

Size \ Distrib	EXP(1)	U(0.0, 2.0)	U(0.9, 1.1)	BIMODAL	DISCR(1)	DISCR(0, 1, 2)
N = 64	0.08	0.08	0.07	0.08	0.06	0.07
N = 256	0.08	0.08	0.08	0.09	0.07	0.09
N = 1024	0.10	0.09	0.10	0.10	0.08	0.10
N = 4096	0.10	0.10	0.11	0.11	0.09	0.10