P-TREE STRUCTURES AND EVENT HORIZON: EFFICIENT EVENT-SET IMPLEMENTATIONS

Katerina Asdre

Department of Computer Science
University of Ioannina
GR-45110 Ioannina, GREECE

Stavros D. Nikolopoulos

Department of Computer Science
University of Ioannina
GR-45110 Ioannina, GREECE

ABSTRACT

This paper describes efficient data structures, namely the *Indexed P-tree*, *Block P-tree*, and *Indexed-Block P-tree* (or *IP*-tree, *BP*-tree, and *IBP*-tree, respectively, for short), for maintaining future events in a general purpose discrete event simulation system, and studies the performance of their event set algorithms under the event horizon principle. For comparison reasons, some well-known event set algorithms were also selected and studied; that is, the Dynamic-heap and the *P*-tree algorithms. To gain insight into the performance of the proposed event set algorithms and allow comparisons with the other selected algorithms, they are tested under a wide variety of conditions in an experimental way. The time needed for the execution of the Hold operation is taken as the measure for estimating the average time complexity of the algorithms. The experimental results show that the *BP*-tree algorithm and the *IBP*-tree algorithm behave very well with all the sizes of the event set and their performance is almost independent from the stochastic distributions.

1 INTRODUCTION

In a discrete event simulation system an *event* (or *future event*) is a collection of actions that are scheduled to be executed in a specific simulation time called *event time*. In such a system events are kept in objects known as *event notices* and maintained in a data structure known as *event set*. An event notice is represented by a record with two fields, t and a, where t is the scheduled time for its occurrence, and a is the activity which is scheduled in time t (Fishman 1973, Mitrani 1982).

In a discrete-event simulation system based on the next-event time-advance approach, the next-event time-advance mechanism is responsible for the simulation clock; it initializes the simulation clock, and then it determines the event times of future events. The simulation clock is then advanced to the event time of the earliest event known as *next event* (i.e., the event with the minimum event time) and the system state is updated to account for the occurrence of this event. When the next event occurs, it is removed from the event set and the simulation clock is advanced to the time of the next event. The processing of this event may lead to the generation and scheduling of additional (new) future events. A new event is scheduled when its event time t becomes known. Then, an event notice is created and inserted into the event set in such a way that it is ensured that this event will occur at the scheduled time t. This type of simulation approach is referred to as *discrete event-driven simulation.*

The responsibility for the execution of these operations in a discrete event-driven simulation is due to an algorithm which is known as an *event set algorithm* (or *event scheduling algorithm*); that is, it

- scans the event set to determine the proper insertion position for the new event,
- removes the next-event from the event set, and
- advances the simulation clock to the time of the next-event.

It is obvious that being able to repeatedly select the event notice from the event set that has the minimum event time is essential. If all of the event notices in this event set are known in advance, and their event times remain unchanged, then the problem of determining the next event and updating the simulation clock is easily solved by sorting the event notices and retrieving them in order. In the simulation process discussed above, however, it is often necessary to insert new event notices into the event set as other events are being processed. This leads to the following set of priority queue operations:

- insert a new event notice into the event set (in a proper position according to its event time),
- find the event notice with the minimum event time, and

- remove the event notice with the minimum event time from the event set.

The above priority queue operations are the most frequent operations required by a discrete event simulation system and they are involved in any event scheduling algorithm. Thus, it is clear that the main factor that affects the efficiency of an event scheduling algorithm is the structure of the event set.

The most important requirements of an event scheduling algorithm are speed of operation and storage economy. Many researchers have extensively studied this field and presented both analytical and empirical results concerning the time and space performance of many event scheduling algorithms. They use different data structures for the simulation of the event sets; that is, linear lists, special kinds of trees, time-indexed lists, two-level structures and many other. Moreover, they use different techniques for the operations performed by the scheduling algorithms; see, (Brown 1988, Franta and Maly 1975, Franta and Maly 1977, Jonassen and Dahl 1975, Kaplan, Shafrir, and Tarjan 2002, McCormack and Sargent 1981, Reeves 1984, Tan and Thng 2000) for an exposition of the main results.

The data structures used for the simulation of the event set can generally be classified under three types; that is, lists, tree structures and multi-lists. Lists are structures that are based on the simple linear list. They include doubly linked lists, indexed lists (Nikolopoulos and MacLeod 1993), SPEEDES Queue which is based on the event horizon technique (Steinman 1992, Steinman 1994, Steinman 1996) and many other. Trees are structures that are based on the simple binary tree, and include binary heaps (Andreou and Nikolopoulos 1998, Franta and Maly 1975, Franta and Maly 1977, Hwang and Steyaert, to appear), skip lists (Nikolopoulos and MacLeod 1993), priority trees (Jonassen and Dahl 1975, Lewis and Denenberg 1991, Nikolopoulos and MacLeod 1993) which are studied here as well. Finally, multi-lists are structures that are the result of a combination of several types of lists. This is done in order to combine the merits of two structures that may not perform as well when implemented separately. Such structures are the calendar queue and the SNOOPY calendar queue (Tan and Thng 2000).

This paper describes efficient data structures, namely the *Indexed P-tree*, *Block P-tree*, and *Indexed-Block P*-tree (or *IP*-tree, *BP*-tree, and *IBP*-tree, respectively, for short), for the simulation event set. All the structures, combine the advantages of both the *P*-tree and the static representation of the list. The combination of the *P*-tree and the list provides efficient date structures for the simulation event set in the case where the event horizon technique is applied. The main feature of each of our event set algorithm is the efficiency of the merging process in the event horizon technique; that is, the process of sorting the event notices of the secondary queue and inserting them back into the event set. We point out that, in the horizon technique the most time consuming operation performed by the event set algorithm is the merging process of the secondary queue back into the main event set.

To gain insight into the performance of the *IP*-tree, the *BP*-tree and the *IBP*-tree, and allow comparisons with other selected algorithms (i.e., Dynamic-heap and *P*-tree), they are coded and tested under a wide variety of conditions in an experimental way. The objective was to estimate the average complexity of 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 scheduled time, we estimate the time expected to be needed for the execution of the Hold procedure (or Hold model).

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. 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 of showing the advantages and limitations of each algorithm. The parameter N defines the notion of the *small* and *large* event sets. Tests were performed with values of N from 64 (small event set) to 262144 (large event set). This range is representative of actual simulations and the behaviour of the algorithms for $N > 262144$ can be extrapolated from the results.

The results of this work show that the *IP*-tree algorithm combines time performance, storage economy and simplicity of coding. The *BP*-tree and the *IBP*-tree algorithms outperform the *IP*-tree algorithm, and the *BP*-tree algorithm has a slightly better performance than the *IBP*-tree algorithm.

The paper is organized as follows. Section 2 presents the main features of the Hold model and the event horizon technique. Section 3 describes the *P*-tree structure which is the structure that our approach is based on. The *IP*-tree, the *BP*-tree and the *IBP*-tree structures are described in Sections 4, 5 and 6, respectively. An experimental evaluation of the algorithms is presented in Section 7, where we also compare the performance of the algorithms. Finally, Section 8 concludes the paper with a summary of our results.

2 HOLD AND EVENT HORIZON

As already mentioned, the two basic operations performed on the event set by an event set algorithm are

(i) insertion of a new event notice into event set, and (ii) determination and deletion of the notice of the next event. A standard metric for comparison of the performance of an event set algorithm is the time required for a Hold operation, which combines both insertion and deletion operations (Andreou and Nikolopoulos 1998, Franta and Maly 1975, Law and Kelton 2000, Mitrani 1982). Under the Hold model, event notices are repeatedly deleted and then re-inserted with a randomly reduced priority; this sequence of operations is known as a Hold operation. The hold operation works as follows:

(1) determine and remove the event notice with the minimum event time T_{min} from the event set; that is, the current notice,

(2) increase the event time value of the current notice by T, where T is a random variate distributed according to some distribution $F(t)$, and

(3) re-insert the new notice back into the event set;it now has $T_{new} = T_{min} + T$ event time.

The Hold model has two parameters: N, the number of notices in the event set, and F, the distribution used to determine the time an inserted event will occur. Thus, the model allows the average combined time for insertion and deletion to be measured as a function of the size of the event set and the stochastic distribution.

The event horizon is a fundamental concept that applies to both parallel and sequential discrete event simulations (Rao and Kumar 1988, Steinman 1992, Steinman 1994, Steinman 1996). Using event horizon one can improve the performance of several event sets; that is, priority queue data structures such as linked lists and various binary trees.

In order to exploit the event horizon for event set management algorithms, it is assumed that as new events are generated they are not inserted into the main priority queue data structure immediately; they are collected in an unsorted temporary (secondary) queue in such a way that one can always track the event with the earliest event time. As a result, when the event to be processed happens to be in the secondary queue, the queue is sorted and then it is "merged" back into the main priority queue.

The secondary queue is most frequently a linked list, providing the advantage of inserting a new event in constant time since the list is kept unsorted; it is sorted just before the merging process. Merging the two data structures, however, is not always a simple process. The main priority queue (event set) itself may be a very complicated data structure.

3 P-TREE

A *Priority-tree* (or *P-tree*) is either empty or it is a sorted, non-increasing sequence of nodes, the "left path", such that to each node of the left path except the last one, is associated a P-tree (possibly empty), the "right subtree". The nodes of the right subtree associated with a node x on the left path, are ranked between x and the left successor of x (Jonassen and Dahl 1975, Lewis and Denenberg 1991, Nikolopoulos and MacLeod 1993).

Neglecting node values, a binary tree is a P-tree if and only if each node having a right successor also has a left successor. The terminal node on the leftmost path is the element with the smallest key value. In order to insert a new element x into a P-tree T the algorithm P-insert below is applied recursively.

P-insert x into T:

1. If $T = \emptyset$ or $x.v \geq T.v$, let x be the new root and T its left subtree;

2. Otherwise search down the left path of T for the first node y, if any, such that $y.v \leq x.v$;

   2.1 If none, append x as the new left leaf;

   2.2 Otherwise $y.v \leq x.v \leq z.v$, where z is the predecessor of y ($y = z.\ell$). P-insert x into the right subtree of z;

where x, y and z denote nodes, $u.\ell$ and $u.v$ denote the left subtree and the node value of u, respectively.

The detection of the event notice with the earliest time value can be performed in constant time provided that there is an additional pointer to the terminal notice on the left path. After the removal of this notice, the last right subtree, if it is not empty, is appended to the left path.

4 THE INDEXED P-TREE

An *Indexed P-tree* (or *IP-tree*) consists of a tree structure, the P-tree, and a static representation of a list structure, the I-list. The elements of the I-list point into specific event notices in the P-tree; see Figure 1. Using the event horizon technique, when the event horizon is crossed (i.e., when the event to be processed happens to be in the secondary queue), the secondary priority queue is sorted and then it is merged back into the main priority queue (P-tree).

We next describe the main operations performed in a discrete event simulation system using the IP-tree for the simulation of the event set.

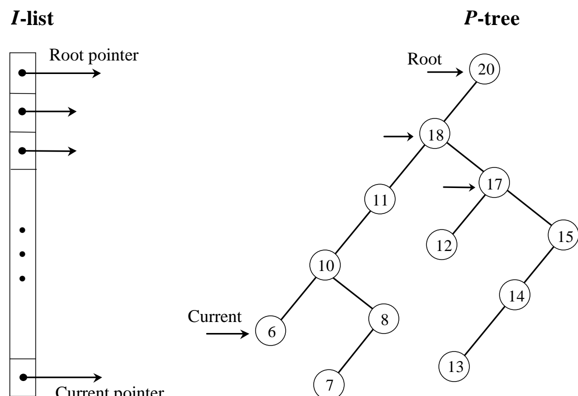

Figure 1: An *IP*-tree structure; it consists of a list of pointers, called I-list, and a P-tree.

(i) **Insert operation:** According to the event horizon technique, a new event notice is inserted in the secondary queue, which is a linked list structure. As the secondary queue is kept unsorted, the insertion of a new event notice can be completed in constant time.

(ii) **Delete operation:** The deletion of the current event notice from the *IP*-tree structure can be implemented in constant time as it only involves deleting the current event notice from a P-tree structure.

(iii) **Merge operation:** Suppose that the event notices of the secondary queue have to be inserted into a standard P-tree structure. To this end, for each event notice the P-tree event set algorithm scans the whole P-tree, starting each time from the root of the tree, in order to determine its proper insertion position. The *IP*-tree event set algorithm takes advantage of the fact that the event notices in the secondary list are sorted in decreasing order and the I-list determines some specific subtrees of the P-tree; recall that every subtree of a P-tree is a P-tree itself. Thus, for each event notice the *IP*-tree algorithm scans the I-list and determines the proper insertion subtree. Then, it proceeds as the P-tree algorithm and completes the insertion operation. Thus, in order to insert the event notices of the secondary list into the *IP*-tree there is no need to scan the whole P-tree for each notice, meaning that the P-insert operation, as it was described before, is not necessary to start from the root of the P-tree.

In the merging process, some of the subtrees that are not scanned during an insertion operation will not be scanned by the next insertion operation either, as the time-value of the event notice of the second operation is less or equal than the time-value of the event notice of the first operation. Taking advantage of this knowledge, the I-list is constructed in order to make the merging operation more efficient. In particular, if an event notice, say b, is the next event to be inserted into the P-tree and the event notice which was last inserted, say a, had a greater event time, we do not need to check a notice, say c, that a was compared with and was found to be less than c. Thus, we need to keep pointers to the event notices of the P-tree that a was compared with and then moved to a right subtree; see Figure 1. We shall call these specific notices I-notices. In the *IP*-tree structure the pointers which point into the I-notices are simply the elements of the I-list. Note that the first I-notice is the root of the P-tree and the last one is the current event notice; that is, the leaf node on the leftmost path of the P-tree.

An example of an *IP*-tree structure is presented in Figure 1. Let a be the last event notice which has been inserted into the P-tree and let 14 be its event time. The pointers of the I-list were pointed at notices with event times 18 and 17. Let b be the next event notice which has to be inserted into the P-tree and let 13 be its event time. Then, the time of the notice b is compared only with the times of the left children of the I-notices. Thus, a search is performed on the I-list and the I-notice that its left child has the greatest event time which is less than the time of b is determined; let c be such an I-notice. Then, the *IP*-tree algorithm P-inserts the notice b into the P-tree rooted at c. Recall that every subtree of a P-tree is also a P-tree.

5 THE BLOCK P-TREE

The *Block* P-tree (or *BP*-tree) structure is a P-tree that consists of nodes containing an array of an initially fixed number of elements, say S, which we call supernodes. The elements of every supernode are kept sorted in increasing order and the P-tree property is applied on the event with the earliest event time of each supernode. In other words, the position of a supernode in the *BP*-tree is determined by the earliest event time that it contains.

Inserting a new event notice is a very simple process. As the *BP*-tree algorithm takes advantage of the event horizon technique, the new event notice can be inserted in constant time in the secondary priority queue which is a static representation of the list structure.

The deletion of the event notice with the earliest event time is quite simple as well. The current supernode, that is the supernode containing the event

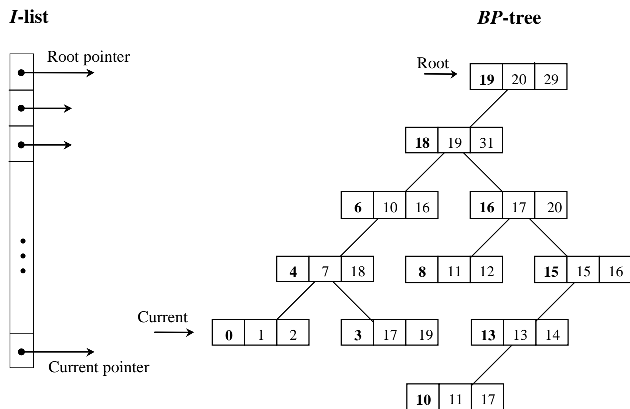

Figure 2: An *IBP*-tree structure; it consists of an *I*-list, and a *BP*-tree with $S = 3$.

notice with the earliest event time, is easily tracked since there always exists a pointer pointing to it, and also since the current event notice can be located in constant time. After deleting the current event notice, the *BP*-tree may need to be updated. Thus, if the new minimum element that the current supernode contains is the minimum of all notices of the *BP*-tree, the deletion operation is completed. Otherwise, the supernode does not contain the current event notice and thus it is reinserted in the *BP*-tree according to the value of the minimum element that it contains. After the completion of this process, the new current supernode contains the current event notice.

Using the event horizon technique, when the event horizon is crossed, the secondary priority queue, which is a static representation of the list structure, is sorted in increasing order. Then, the secondary queue forms a single supernode which is then reinserted back into the main priority queue (*BP*-tree). The advantage of this implementation is that only one supernode has to be inserted into the *BP*-tree. Note that, the number of the elements of each supernode can exceed the value of S, as the secondary queue can consist of more than S elements when the event horizon is crossed.

6 THE INDEXED BLOCK P-TREE

The *Indexed-block P*-tree (or *IBP*-tree) structure is based on the *IP*-tree and the *BP*-tree structures; it consists of an *I*-list and a *BP*-tree. Recall that the *I*-list is a static representation of the list structure and its elements point into specific event notices of the *BP*-tree; see Figure 2.

The insertion operation is a very simple process, since the *IBP*-tree algorithm takes advantage of the event horizon technique; that is, the new event notice can be inserted in constant time in the secondary priority queue which is a static representation of the list structure.

The deletion of the event notice with the earliest event time is quite simple as well and similar to the deletion operation of the *BP*-tree algorithm. The current supernode is tracked in constant time as there is a pointer variable pointing to it. Since the elements of the supernodes are sorted, the current event notice can be located in constant time. After extracting it, the *BP*-tree may need to be updated. This operation is similar to that performed by the *BP*-tree algorithm. Note that, despite the fact that the deletion operation may result to supernodes having less than S elements, it is essential that the supernodes cannot consist of more than S elements.

Using the event horizon technique, when the event horizon is crossed, the secondary priority queue, which is a static representation of a list structure, is sorted in increasing order and then it is merged back into the main priority queue (*BP*-tree).

Let us describe the merging process of the *BP*-tree structure and the secondary queue in the event horizon technique. The event notices of the secondary queue, after being sorted, form arrays (supernodes) that contain S elements. Thus, if the secondary queue contains M elements, M/S supernodes have to be reinserted at the *BP*-tree; each supernode contains S event notices, except probably from the last one. Recall that the proper insertion position of each supernode is determined according to the value of the earliest event time that it contains, say t_{min}. In order to complete the merging process, we take advantage of the *IP*-tree algorithm and the *I*-list. Thus, for each supernode, the *IBP*-tree algorithm scans the *I*-list and determines the proper insertion subtree. Then the algorithm proceeds as the *P*-tree algorithm and completes the insertion operation. Note that the supernodes are inserted in the *BP*-tree in decreasing order according to the value of the earliest event time that each supernode contains.

An example of a *IBP*-tree structure is presented in Figure 2. After deleting the current event notice, which in our example has value equal to 0, the *BP*-tree is not updated since the last supernode of the leftmost path of the tree still contains the current event notice, which has now value equal to 1.

7 AN EXPERIMENTAL EVALUATION OF THE ALGORITHMS

The main motivation for the empirical studies performed so far comes from the fact that most of the

Table 1: The Six Distributions

(A) Unimodal

- EXP: Negative exponential (mean 1).
- U02: Uniform distribution over the interval [0, 2].
- U09: Uniform distribution over [0.9, 1.1].

(B) Bimodal

- BIM: 0.9 probability - uniform over [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

- D1: T is constant with value of unity.
- D012: T is assigned the values 0, 1 or 2 with equal probabilities.

theoretical performance bounds associated with the different event set algorithms hide significant constant factors. In addition, these results are usually expressed using different concepts such as expected case, worst case and amortized case bounds. Given the number of alternatives for implementing the event set and the need for solutions that are efficient in practice, the empirical studies have then arisen as an effective tool to evaluate the performance of an event set algorithm.

7.1 Test Conditions

Most of the research performed to date uses the Hold procedure to estimate the average time complexity of an event set algorithm. The time needed for the execution of the Hold operations is the measure for estimating the average time complexity. Obviously, the data structure chosen to simulate the event set as well as the size of the event set affect the processor time required for the Hold operations (insertion and deletion). Tests were performed for $N = 2^k$, $k = 6, 7, \ldots, 18$, where N is the size of the event set; that is, the number of event notices in the event set.

A crucial step in designing the tests lies in the selection of the stochastic distribution which provides the event time T; that is, the parameter for the Hold procedure that determines how long an event notice remains in the event set. Six distributions have been chosen because they differ in their characteristics and reveal the advantages and the disadvantages of an algorithm; see Table 1. Each test includes the following operations:

(1) generate N event notices with each one having event time that is generated by the distribution F and insert them into the event set.

(2) without counting time, execute 1.6×10^6 times the Hold procedure with the distribution F.

Table 2: Dynamic-heap Algorithm: Test Results

N	**U02**	**U91**	**EXP**	**BIM**	**D1**	**D012**
64	5.11	5.13	5.42	6.68	5.10	5.13
256	5.85	5.90	6.20	7.42	5.83	5.87
1024	6.60	6.58	6.88	8.14	6.51	6.57
4096	7.41	7.42	7.71	8.95	7.29	7.37
16384	8.36	8.35	8.76	9.95	8.03	8.11
65536	9.35	9.33	9.62	10.93	8.83	8.94
262144	10.53	10.54	10.89	12.12	9.54	9.74

Table 3: P-tree Algorithm: Test Results

N	**U02**	**U91**	**EXP**	**BIM**	**D1**	**D012**
64	3.27	2.67	3.78	7.87	2.33	3.58
256	4.02	2.99	4.48	16.15	2.33	6.80
1024	5.33	3.50	5.42	31.42	2.33	17.77
4096	7.82	4.42	6.41	58.61	2.35	73.14
16384	12.66	6.10	7.83	102.90	2.36	308.95
65536	22.00	9.26	9.52	141.88	2.39	1253.46
262144	54.41	20.78	13.69	280.26	2.42	4995.45

(3) execute 1.6×10^6 Hold operations and count the total processor time (CPU time) needed to complete them.

Operation (1) initializes the system while operation (2) allows it to reach a steady state. Operation (3) yields a measure of the complexity of the tested algorithms. The algorithms were coded in C programming language and the experimental results were taken from Sun-Blade-1000, 2×750 MHz Ultrasparc-III processors (8MB cache), 512MB RAM.

The *IP*-tree, the *BP*-tree and the *BIP*-tree algorithms were coded and run to collect evidence of their performance under realistic conditions. For comparison reasons, well-known event set algorithms were also coded and run under the same conditions; that is, the Dynamic-heap and the P-tree algorithms. To gain insight into the performance of the proposed algorithms and allow comparisons with the other event set algorithms, they were tested under a wide variety of conditions in an experimental way. The experimental results for each algorithm (that is the time in seconds needed to complete each algorithm) are represented in the form of tables and graphs; see, Tables 2–6 and Figures 3–8.

7.2 Dynamic-heap and P-tree: Hold model

As expected, the Dynamic-heap algorithm provides a very good time performance. The performance results are given in Table 2. What is observed is the expected logarithmic behavior of a heap data structure and

the fact that the time performance of the event set algorithm is almost the same with all the distributions. We note that the Static-heap algorithm has the same performance.

One can easily observe (see Table 3), that the performance of the P-tree is not as good as the performance of the heap algorithm. Its CPU times increase with the variance of the scheduling distribution. It is remarkable that the P-tree is efficient under constant values (D1 distribution). This performance was expected because each new event notice becomes the new root of the P-tree, which in this case is a sorted linked list, and thus the new event is inserted in constant time. Its performance is extremely worst with the discrete D012 distribution. Furthermore, the P-tree algorithm becomes even more inefficient as the size of the event set increases. The experimental results showed that the performance of the P-tree algorithm cannot be improved by applying the event horizon technique.

7.3 IP-tree, BP-tree and IBP-tree: Event Horizon

The experimental results of the performance of the IP-tree algorithm are presented in Table 4. These show the superiority of the IP-tree compared to the P-tree algorithm and its excellent performance with all the sizes of the event set and all the stochastic distributions. Specifically, we observe that the CPU time for the D012 distribution is extremely decreased compared with the results taken by the P-tree algorithm. In addition, one can observe that the CPU times for all the distributions are almost the same.

The experimental results of the performance of the BP-tree algorithm are presented in Table 5. The results show the superiority of the algorithm compared to the heap algorithm, and also to the P-tree and IP-tree algorithms. Furthermore, its performance is slightly better than the performance of the IBP-tree, apart from the Exponential and the discrete D012 distributions when $N > 65536$. Note that the value of the parameter S is equal to the size of the event set; that is $S = N$. Recall that the parameter S determines only the initial size of the supernodes because the size changes as the secondary queue becomes a supernode every time the event horizon is crossed. The experimental results showed that the algorithm performs better when the BP-tree has initially only one supernode which contains N elements; that is when $S = N$.

Table 6 presents the experimental results of the performance of the IBP-tree algorithm. Its excellent performance, regardless of the size of the event set or the stochastic distribution, show the superiority of the algorithm over the IP-tree and the heap algorithms. The IBP-tree algorithm outperforms the IP-tree algorithm because it takes advantage of the properties of the latter and, in addition, the size of the event set can be considered to be smaller as the event notices form supernodes. Thus, if the size of the event set is equal to N, the IP-tree algorithm produces a P-tree containing N nodes, while the IBP-tree algorithm can produce a BP-tree much smaller, having N/S nodes, if the value of S is sufficiently large. The experimental results show that the IBP-tree algorithm has the best performance when $S \approx 3000$. Consequently, when $N < S$ the IBP-tree algorithm behaves as the BP-tree algorithm.

Table 4: IP-tree Algorithm: Test Results

N	**U02**	**U91**	**EXP**	**BIM**	**D1**	**D012**
64	4.04	3.81	4.39	5.26	3.46	3.72
256	4.46	4.17	4.88	6.37	3.66	4.01
1024	4.89	4.52	5.36	7.00	3.88	4.32
4096	5.43	5.02	5.98	7.79	4.13	4.72
16384	6.12	5.65	6.76	8.36	4.48	5.28
65536	6.86	6.21	7.50	8.52	4.74	5.72
262144	9.08	8.50	9.97	9.96	5.15	7.78

Table 5: BP-tree Algorithm: Test Results

N	**U02**	**U91**	**EXP**	**BIM**	**D1**	**D012**
64	3.92	3.45	4.38	4.75	3.26	3.42
256	4.44	3.78	4.96	5.29	3.47	3.67
1024	4.98	4.13	5.58	6.15	3.66	3.92
4096	5.60	4.55	6.30	6.86	3.95	4.23
16384	6.30	5.08	7.09	7.41	4.24	4.66
65536	7.10	5.57	8.02	7.99	4.46	5.03
262144	8.52	7.69	9.94	9.34	4.77	6.74

Table 6: IBP-tree Algorithm: Test Results

N	**U02**	**U91**	**EXP**	**BIM**	**D1**	**D012**
64	4.07	3.50	4.58	5.76	3.41	3.54
256	4.52	3.81	5.08	6.59	3.58	3.73
1024	5.03	4.15	5.65	7.39	3.76	3.96
4096	5.65	4.59	6.33	7.84	4.04	4.25
16384	6.35	5.11	7.14	8.05	4.35	4.68
65536	7.14	5.59	8.06	8.25	4.56	5.01
262144	8.66	7.66	9.91	9.45	4.91	6.63

7.4 A Comparison of the Algorithms

The experimental results show that the IP-tree algorithm has an extremely better performance than the P-tree algorithm. The latter becomes very inefficient as the size of the event set increases, especially with the D012 distribution. The BP-tree and IBP-tree al-

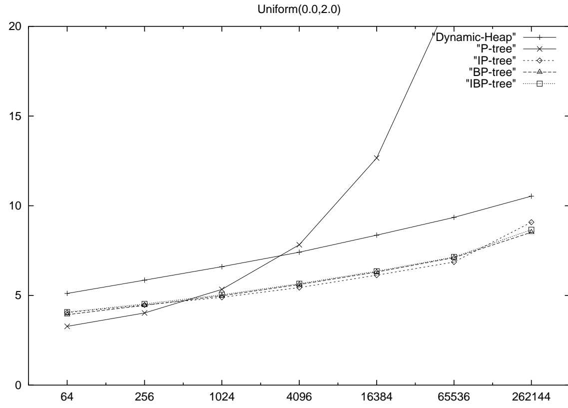

Figure 3: Uniform(0.0,2.0) Distribution

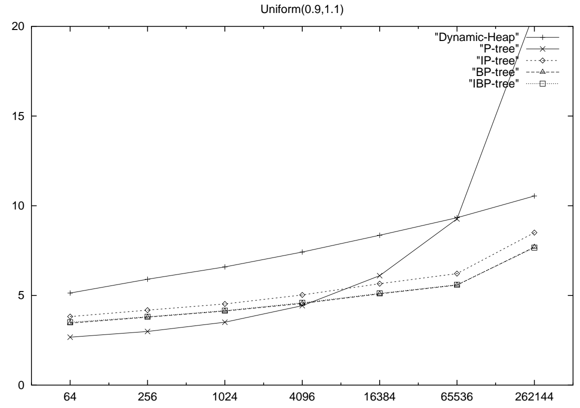

Figure 6: Uniform(0.9,1.1) Distribution

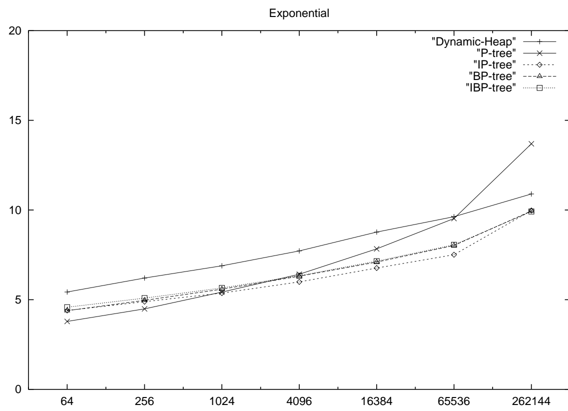

Figure 4: Exponential Distribution

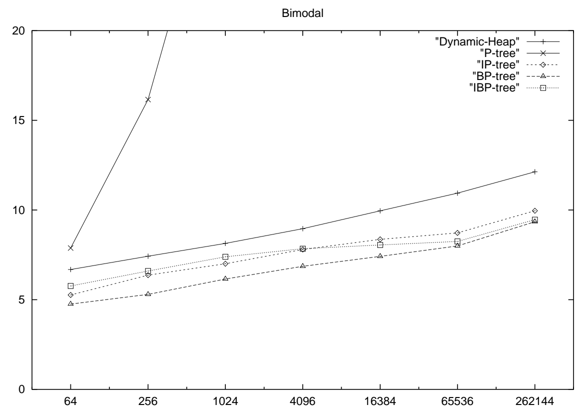

Figure 7: Bimodal Distribution

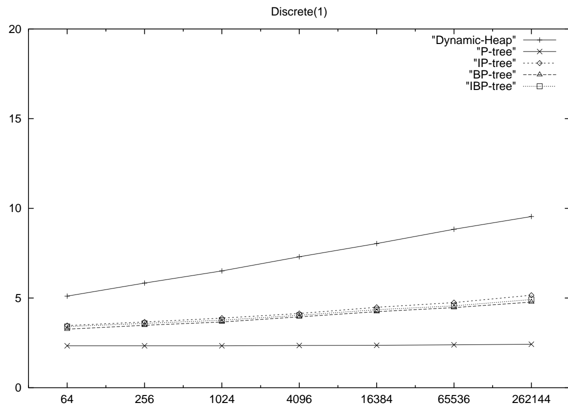

Figure 5: Discrete(1) Distribution

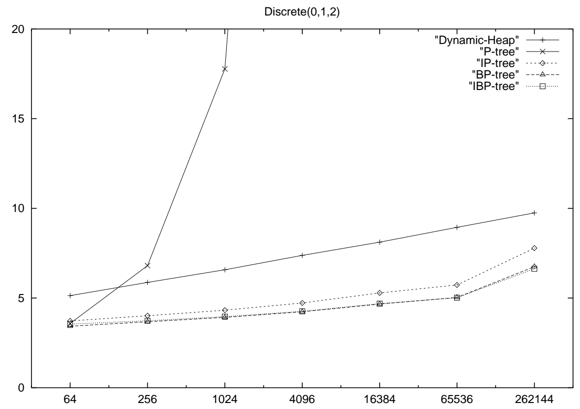

Figure 8: Discrete(0,1,2) Distribution

gorithms are even more efficient than the *IP*-tree algorithm as their performance is excellent regardless of the size of the event set or the distribution that is used.

What is also remarkable is the fact that the *BP*-tree and the *IBP*-tree algorithms provide results which are better even than the results of the well known efficient Dynamic-heap algorithm. The latter performs, as expected, better than the *P*-tree algorithm regardless of the distribution which is used. Furthermore, the *BP*-tree and the *IBP*-tree algorithms outperform all the

other algorithms and their superiority can easily be concluded. Figures 3–8 present the performance of each one of the algorithms under the six distributions.

We would like to comment on the logarithmic behaviour of the *BP*-tree and the *IBP*-tree algorithms. One can easily observe from Figures 3–8 that the two algorithms behave like the Dynamic-heap algorithm. Furthermore, the performance of the *IP*-tree algorithm resembles the performance of the *P*-tree algorithm.

What should also be pointed out is that the event horizon technique, when applied to the Dynamic-heap algorithm or the *P*-tree algorithm, does not result to a better performance. Applying the event horizon principle to the heap algorithm involves using a heap structure (static representation) as a main event set and a list data structure (unsorted array) as a secondary event set. When the minimum event time is found to be in the secondary list, its elements are merged back into the main priority queue data structure (merge operation). The array is kept unsorted because experimental results showed that the performance of the Static-heap is not improved in the case where the secondary list is sorted either in increasing or decreasing order. Furthermore, it was observed that the event horizon technique does not affect the performance of the Dynamic-heap algorithm; that is, the performance of the algorithm with the event horizon technique is almost the same as without it.

In the *P*-tree algorithm the secondary data structure is an unsorted linked list. When the next-event to be processed (event notice with the minimum event time) happens to be in the secondary list, the latter is sorted in an increasing order and its elements are placed back into the main event set. The experimental results showed that when we apply event horizon the CPU times taken by the *P*-tree algorithm are slightly increased for all the distributions except for the D012 distribution.

8 CONCLUDING REMARKS

The *P*-tree structures proposed in this paper could usefully replace the classic *P*-tree structure, as well as the heap structure, for the simulation event set in a general purpose discrete event simulation system. The processor time obtained with the *IP*-tree, the *BP*-tree and the *IBP*-tree algorithms is relatively insensitive to variations in the scheduling distributions or the number of event notices in the event set, and points to their superiority over the *P*-tree structure, and also over the Dynamic-heap. Our proposed structures provide time efficiency, size flexibility and space economy.

Future work might involve how the *BP*-tree or *IBP*-tree algorithms can be efficiently parallelized. Furthermore, it would be interesting to study the performance of algorithms that use other tree-like data structures under the event horizon technique and/or the *I*-list technique.

In closing, we point out that the results of this work prompts us to suggest the *BP*-tree and the *IBP*-tree as efficient data structures for the simulation event set in a general purpose discrete event simulation system.

REFERENCES

Andreou, M., and S. D. Nikolopoulos. 1998. An efficient data structure for maintaining future events in a discrete-event simulation system. in: *Recent Advances in Information Science and Technology*, World Scientific: 11–22.

Brown, R. 1988. A fast O(1) priority queue implementation for the simulation event set problem. *Comm. of the ACM* 31: 1220–1227.

Fishman, G. S. 1973. *Concepts and methods in discrete event digital simulation.* John Wiley & Sons.

Franta, W. R., and K. Maly. 1975. A comparison of heap and the TL structure for the simulation event set. *Comm. of the ACM* 21: 873–875.

Franta, W. R., and K. Maly. 1977. An efficient data structure for the simulation event set. *Comm. of the ACM* 20: 569–602.

Hwang, H. K., and J. M. Steyaert, 2002. On the number of heaps and the cost of heap construction. in: *Mathematics and Computer Science II*, Birkhauser Verlag, Basel, 294-310.

Jonassen, A., and O. J. Dahl. 1975. Analysis of an algorithm for priority queue administration. *BIT* 15: 409–422.

Kaplan, H., N. Shafrir, and R. E. Tarjan. 2002. Meldable heaps and boolean union find. *Proc. 34th ACM Symp. on Theory of Computing (STOC'02)*, 573–582.

Law, A. M., and W. D. Kelton. 2000. *Simulation modeling and analysis.* 3rd ed. McGraw-Hill Inc.

Lewis, R., and L. Denenberg. 1991. *Data structures & algorithms.* New York: Harpel Collins Publishers.

McCormack, W. M., and R. G. Sargent. 1981. Analysis of future event set algorithms for discrete event simulation. *Comm. of the ACM* 24: 801–812.

Mitrani, I. 1982. *Simulation techniques for discrete event systems.* Cambridge University Press.

Nikolopoulos, S. D., and R. MacLeod. 1993. An experimental analysis of event set algorithms for discrete event simulation. *Microprocessing and Microprogramming* 36: 71–81.

Rao, V. N., and V. Kumar. 1988. Concurrent access of priority queues. *IEEE Trans. Comput.* 37: 1657–1665.

Reeves, C. M. 1984. Complexity analyses of event set

algorithms. *The Computer Journal* 27: 72–79.

Steinman, J. S. 1992. SPEEDES: A multiple synchronization environment for parallel discrete-event simulation. *Inter. J. Computer Simul.* 2: 251–286.

Steinman, J. S. 1994. Discrete-event simulation and the event horizon. *Proc. 10th Workshop Parall. and Distrib. Simul. (PADS'94)*, 39–49.

Steinman, J. S. 1996. Discrete-event simulation and the event horizon Part 2: Event list management. *Proc. 8th Workshop Parall. and Distrib. Simul. (PADS'96)*, 170–178.

Tan, K. L., and L. J. Thng. 2000. SNOOPY calendar queue. In *Proceedings of the 2000 Winter Simulation Conference*, 487–495.

AUTHOR BIOGRAPHIES

KATERINA ASDRE received the B.Sc degree (with Honours) and the M.Sc degree in Computer Science from the University of Ioannina, Greece, in 1999 and 2001, respectively. She is currently a Ph.D student in the Department of Computer Science at the University of Ioannina, Greece. Her research interests are in parallel algorithms, data structures and algorithms, graph algorithms, simulation techniques, discrete event systems. Her e-mail address is <katerina@cs.uoi.gr>.

STAVROS D. NIKOLOPOULOS received the B.Sc degree (with Honours) in Mathematics from the University of Ioannina, Greece, in 1982, the M.Sc degree in Computer Science from University of Dundee, Scotland, in 1985, and the Ph.D degree in Computer Science from the University of Ioannina, in 1991. He was a Research Assistance in the Saclant Undersea Research Centre, Italy, in 1990-91. From 1992 to 1996 he was a Lecturer in the Department of Computer Science, University of Cyprus, Nicosia. In 1996 he joined the Department of Computer Science, University of Ioannina, Greece, where he is currently an Associate Professor. His research interests focus on parallel algorithms, data structures and algorithms, graph theory, graph modelling, combinatorial and graph algorithms, combinatorial mathematics, simulation techniques, discrete event systems. His e-mail address is <stavros@cs.uoi.gr>, and his web page is <www.cs.uoi.gr/stavros>.