André Schiper (Ed.)

Distributed Algorithms

7th International Workshop, WDAG '93
Lausanne, Switzerland, September 27-29, 1993
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
D-76131 Karlsruhe, Germany

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Volume Editor

André Schiper
Ecole Polytechnique Fédérale de Lausanne
Département d'Informatique
Laboratoire de Systèmes d'Exploitation
CH-1015 Lausanne, Switzerland

CR Subject Classification (1991): F.1, D.1.3, F.2.2, C.2.2, C.2.4, D.4.4-5

ISBN 3-540-57271-6 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-57271-6 Springer-Verlag New York Berlin Heidelberg

Printed in Germany

Typesetting: Camera-ready by author
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Preface

The SEVENTH INTERNATIONAL WORKSHOP ON DISTRIBUTED ALGORITHMS (WDAG 93) was held September 27-29 in Lausanne, Switzerland (more precisely in *Les Diablerets*, a village located near Lausanne). The workshop followed six successive workshops held in Ottawa (1985, proceedings published by Carleton University Press), Amsterdam (1987, proceedings published by Springer Verlag, LNCS 312), Nice (1989, LNCS 392), Bari (1990, LNCS 484), Delphi (1991, LNCS 579) and Haifa (1992, LNCS 647). The WDAG provides an international forum for the presentation of new research results and the identification of future research directions in the area of distributed algorithms.

Submissions were solicited in all areas of distributed algorithms and their applications, including distributed algorithms for control and communication, fault-tolerant distributed algorithms, network protocols, algorithms for managing replicated data, protocols for real-time distributed systems, issues of asynchrony, synchrony and real-time, mechanisms for security in distributed systems, techniques for the design and analysis of distributed algorithms, distributed database techniques, distributed combinatorial and optimization algorithms, and distributed graph algorithms.

A total of 72 papers were received within the submission deadline (33 submissions from Europe, 29 from North America, 6 from the Middle East, 3 from the Far East, and 1 from Australia). The Program Committee wishes to thank all the authors who submitted papers for consideration. Out of the 72 submissions the Program Committee was able to select the 22 papers appearing in these proceedings (6 from Europe, 13 from North America, and 3 from the Middle East). The selection was based on originality and quality. Relevance of the papers to the field of distributed computing was also considered carefully.

The Program Committee was composed of:

D. Dolev (Hebrew U. and IBM Almaden)	M. Herlihy (DEC CRL)
G. Le Lann (INRIA, Paris)	K. Marzullo (Cornell U. and UC San Diego)
F. Mattern (U. of Saarland)	M. Raynal (IRISA, Rennes)
M. Merritt (AT&T)	P. Spirakis (CTI and Patras U.)
A. Schiper (*chair*, EPF Lausanne and Cornell U.)	J. van Leeuwen (U. of Utrecht)
J. Welch (Texas A&M U.)	S. Zaks (Technion, Haifa)

I wish to thank all the members of the Program Committee and all the referees who assisted them for their careful reviewing carried out within a very short period of time. My thanks go also to Alain Sandoz for his excellent organization of the Workshop, and for his smooth handling of the submitted and accepted papers.

For the first time this year, a tutorial was organized during the Workshop. The title of the tutorial was *Specifications and Algorithms for Fault-Tolerant*

Broadcasts – A Modular Approach. It was presented by Sam Toueg from Cornell University. The idea in combining a tutorial with research papers was to attract young researchers to the Workshop, and hopefully to the field of distributed algorithms. Financial support for the tutorial was provided by the 3ème Cycle Romand d'Informatique, for which we are grateful.

Lausanne, September 1993 André Schiper

List of Referees

Anceaume, E.
Attiya, H.
Bakker, E.M.
Breitbach, T.
Buehler, P.
Charron-Bost, B.
Cidon, I.
Clegg, M.
Dolev, D.
Dolev, S.
Fix, L.
Fuenfrocken, S.
Guerraoui, R.
Haldar, S.
Helary, J.-M.
Herlihy, M.
Herman, T.
Israeli, A.
Le Lann, G.
Lueling, R.
Malki, D.
Marzullo, K.
Matsliach, G.
Mattern, F.
Merritt, M.
Minet, P.
Moran, S.
Papatriantafillou, M.
Plouzeau, N.
Priebe, V.
Rachman, O.
Raynal, M.
Ricciardi, A.
Richter, J.
Sandoz, A.
Schiper, A.
Schwarz, R.
Spirakis, P.
Tampakas, B.
Tan, R.B.
Tel, G.
Tsigas, Ph.
van Leeuwen, J.
Welch, J.
Zaks, S.
Zamsky, A.

Table of Contents

Wait-free synchronization

Shared memory model

Miscellaneous

Fault tolerance

Networks and rings

Miscellaneous

Efficient Wait-Free Implementation of a Concurrent Priority Queue

Amos Israeli[1] and Lihu Rappoport[2]

[1] Faculty of Electrical Engineering, Technion, Israel
[2] Faculty of Computer Science, Technion, Israel

Abstract. We present an efficient wait-free implementation of a concurrent *priority-queue* in the *asynchronous shared memory* computational model. In this model each process runs at different speed and might be subject to arbitrarily long delays. The new implementation is based on the *heap* data structure in which the *Insert* and *DeleteMin* operations are *long* – they take more than one atomic instruction to complete and they leave the heap inconsistent until completed. The previous implementation requires copying the entire data-structure by each processes every time it tries to perform an operation. Consequently its space and time complexity are linear in the number of processes – p and in the size of the data-structure – n. In the new implementation all processes operate directly on the shared copy of the data structure. Its time complexity is $O(p \log n)$ and its space complexity is $O(n)$. Moreover, the new implementation is *effectively parallel*, meaning that all processes can operate effectively on the object, such that the throughput increases as the number of processes increases.

1 Introduction

In this paper we present a wait-free implementation for a concurrent *priority queue* in the asynchronous shared memory model. We show that the new implementation is more efficient then the previously known implementations in *space*, *time* and *processor utilization*. We use the *asynchronous shared memory* computational model. In this model a group of processes communicate via shared memory. Each process runs at a different speed, and might be subject to arbitrarily long delays. A *concurrent object* is a data structure residing in the shared memory and accessible by some of the system processes.

The traditional technique for implementing concurrent objects is by the use of *critical sections*: ensuring that only one process operates on the object at a given time. The use of critical sections in asynchronous systems is problematic in at least two ways:

1. If a process is delayed inside the critical section, all the other processes cannot make any progress.
2. At any given moment only one process can access the object. Thus a more appropriate name for an object implemented using critical sections is a *shared object*, rather than a *concurrent object*.

A concurrent object implementation is *non-blocking* if it always guarantees that *some* process completes an operation in a bounded number of a steps. A concurrent object implementation is *wait-free* if it guarantees that *each* process completes an operation within a bounded number of steps.

1.1 Related Work

The work on concurrent wait-free objects starts with the work of Peterson in [15] and Lamport in [13, 14] on atomic registers, and continued with the work on consensus objects [12, 6, 3, 4]. The work on data structures was initiated by Herlihy in [7] where he defines a hierarchy of concurrent objects such that there is no wait-free implementation of an object using only objects lower in the hierarchy. Herlihy shows that there exist *universal objects* which allow for a wait-free implementation of *any* concurrent object. This result however does not relate to the efficiency of the implementation. Anderson and Woll in [5] show a wait-free implementation for the *Union-Find* problem.

Herlihy in [8] introduces a general method for converting a sequential data structure to a wait-free shared object. He uses the *Load Linked* and the *Store Conditional* universal atomic primitives. As an example he implements a priority-queue using the heap data structure. The basic idea of Herlihy's method is as follows: The shared object is pointed at by a shared pointer. To apply an operation to the shared object, process P_i reads the pointer using *Load Linked* and copies the object to a local copy in the shared memory. Then P_i applies the operation *sequentially* to its local copy, and finally it tries to swing the shared pointer to point to its local copy, using *Store Conditional*. This *Store Conditional* instruction succeeds only if the pointer was not changed by some other process since it was last read by P_i by the *Load Linked* instruction. This method yields a non-blocking implementation, and by using a technique called *operation combining*, it is converted to be wait-free.
The method of [8] has three significant drawbacks:

1. *A large amount of memory is needed* (for the local copies), thus the space complexity of an implementation obtained using this method is at least p times the space complexity of the sequential implementation it is using, where p is the number of the system's processes.
2. *A great deal of copying must be performed*, thus the time complexity of an implementation obtained using this method is at least *linear* in n, the size of the data structure.
3. *At any given moment, only one process can execute effective instructions.* Each process executes operations sequentially on its local copy, which is "locked" from other processes. Of all overlapping trials, only one process succeeds in making its local copy the new version of the concurrent object, while the work done by all other processes is wasted.

Thus any implementation obtained by this method is *inherently sequential*. If execution of some operation on the data structure by a single processor takes time t, then execution of r operations takes time at least $r \cdot t$, regardless of how many processes participate in executing the r operations. At any given moment, at most one process can execute effective instructions. We define an implementation to be *effectively parallel* if it has at least one execution in which execution of r operations take less then $r \cdot t$ time.

Alemany and Felten in [2] reduce the excessive copying and wasteful work in Herlihy's implementation by using information extracted by the operating system to identify faulty or slow processes. By using the operating system as an oracle, the primary problem of the asynchronous model, that one process cannot tell whether another process is halted, is avoided. However, the implementation in [2] is still inherently sequential.

1.2 The Current Work

In this paper we present a wait-free implementation of a concurrent priority-queue using a *heap*. A *priority-queue* supports two operations: *Insert* – adds an item to the priority-queue, and *DeleteMin* – deletes the item with the highest priority from the priority-queue and returns it. In the sequential implementation the *Insert* and *DeleteMin* operations in a heap are *long* – they require more than a single atomic instruction to complete and they leave the concurrent object in an inconsistent state until completed. For example, in an *Insert* operation, the inserted item traverses the data structure, in a sort–like procedure, until it reaches its proper location; as long as the item does not get to its proper location, the heap is inconsistent. Long operations pose a problem in wait-free implementations: a new operation may be started before a previous operation is completed, so the data structure may be inconsistent while more than one operation is in progress.

The new implementation improves upon the implementation of [8] in all three aspects mentioned above:

1. Its space complexity is linear in n, the number of items in the priority queue.
2. Its time complexity is *logarithmic* in n, (though in the worst case it is still linear in p).
3. It is effectively parallel.

The worst case bound for executing a set of r *Insert* and *DeleteMin* operations is at most $O(pr \log n)$ instructions (by all processes together), compared with $O(prn)$ instructions using the implementation in [8] (n is the maximum number of items in the priority-queue during execution of the r operations). The space complexity of the current implementation is $O(n)$ memory locations, compared with $O(pn)$ in [8]. In the new implementation an operation may be started before previous operations are completed while the object is inconsistent. Although the object is not always consistent, the correctness holds.

An important complexity measure is the maximal number of memory words accessed by a single atomic instruction in the implementation. It is easy to see that if a process is allowed to execute read-modify-write instructions that access unlimited number of memory words, there exists a simple and efficient transformation for any sequential object to a wait-free concurrent object. Thus the quality of an implementation should be evaluated also by the maximum number of memory words accessed by a single atomic instruction in that implementation. The new implementation uses atomic primitives that access at most two memory words simultaneously.

The rest of the paper is organized as follows: In section 2 we describe the computational model used. The wait-free implementation of the priority-queue is constructed in two stages. In section 3 we present a non-blocking implementation of a priority-queue. In section 4 we introduce a technique by which the non-blocking implementation of the priority-queue can be made wait-free.

2 The Model

We use the *asynchronous shared memory* computational model. In this model a group of sequential processes (or processors) communicate via shared memory. The processes are *asynchronous* – there is no global clock timing them; each process runs at a different speed, and might be subject to arbitrarily long delays. A process cannot tell whether another process is halted or is running very slowly. All of the instructions executed by the processes are *atomic*, meaning that they seem to be executed in a certain point of time, such that no two instructions are executed at the same moment, and that the instructions can be ordered. An *execution* is an interleaving of the atomic instructions executed by the processes in the system.

A *concurrent object* is a data structure shared by concurrent processes. Each object has a *type*, which defines a set of possible primitive operations and a set of possible values for each operation. The primitive operations provide the only means to manipulate the object. Each object has a *sequential specification* that defines how the object behaves when its operations are invoked one at a time (the sequence of responses to each sequence of allowed operations). The *execution interval* of an operation is the time interval between the operation invocation and the corresponding response. In the sequel, the term *instruction* refers to an instruction in the instruction set of the machine, and the term *operation* refers to an operation defined on an object.

Intuitively, an *implementation* of a concurrent object A, is another concurrent object I, such that the processes in the system cannot distinguish between A and I. A concurrent implementation is said to be *correct* if for any sequence of legal operations, for any execution, it is possible to define an *occurrence time* for each operation, such that the following two conditions hold:

1. The occurrence time of each operation is within its execution interval.
2. When the operations are ordered according to their occurrence times, the sequence of corresponding responses agrees with the sequential specification of the object.

This correctness condition, called *linearizability*, is defined in [10].

A concurrent object implementation is *non-blocking* if it always guarantees that *some* process completes an operation within a bounded number of a steps. A concurrent object implementation is *wait-free* if there exists a positive integer k such that it is guaranteed that a process executes at most k instructions in order to complete any operation defined on the object, regardless of the speed of other processes.

We use the following atomic primitives to access shared variables: *Read*, *Write*, *Load Linked (LL)*, *Validate (VL)*, *Store Conditional (SC)*, *Store Conditional 2 (SC2)* and *Store Conditional & Validate (SC&V)*. Let x and y be shared variables, and let a and b be local variables or values. We then define:

- *Read(x)*: Read the value of x.
- *Write(x,a)*: Write the value a to x.
- *LL(x)*: Read the value of x such that it may be subsequently used in combination with each of *VL*, *SC*, *SC&V* and *SC2*.
- *VL(x)*: If x is not written since the last *LL(x)* instruction executed, return *SUCCESS*, otherwise return *FAILURE*.
- *SC(x,a)*: If x is not written since the last *LL(x)* instruction executed, write the value a to x and return *SUCCESS*, otherwise return *FAILURE*.
- *SC&V(x,a,y)*: If x and y are not written since the last *LL(x)* and *LL(y)* instructions executed respectively, write the value a to x and return *SUCCESS*, otherwise return *FAILURE*.
- *SC2(x,a,y,b)*: If x and y are not written since the last *LL(x)* and *LL(y)* instructions executed respectively, write the value a to x, write the value b to y and return *SUCCESS*, otherwise return *FAILURE*.

These primitives can be implemented using the *transactional memory* scheme, introduced by Herlihy and Moss in [9]. Transactional memory allows to define customized read-modify-write operations that access multiple, independently-chosen words of memory. Primitives *LL* and *SC* are used in [8], *VL* is suggested in [9], *SC2* and *SC&V* are the natural generalization of *SC* and *VL* for the case of accessing two memory words simultaneously.

3 The Non-Blocking Implementation

A priority-queue supports two operations:

Insert – If the priority-queue is not full, adds an item to the priority-queue and returns *SUCCESS*, otherwise returns *FAILURE*.

DeleteMin – If the queue is not empty, deletes the item with the highest priority from the priority-queue and returns that item. If the priority-queue is empty returns *FAILURE*.

We use a *heap* to implement the priority-queue. A detailed description of the heap data structure can be found in [1]. A heap is a complete binary tree, in which each node has a value less than or equal to the values of its sons.

This implies that for every node v, the values of all nodes on the path from v to the root have values that are less than or equal to v's value and that a root of a any sub-tree in the heap is the least of all nodes in that sub-tree. A heap implements a priority-queue sequentially with *Insert* and *DeleteMin* both executed in $O(\log n)$, where n is the number of items in the priority-queue. Lower values correspond to nodes with higher priorities.

The heap is usually implemented as an array $Heap[1..N]$ of ***locations***, where N is the maximum size of the priority-queue. Each location can hold a ***node***. $Heap[1]$ is the ***root*** of the heap. The ***right son*** of the node in $Heap[i]$ is in the node in $Heap[2i+1]$ and the ***left son*** is the node in $Heap[2i]$. The ***parent*** of the node in $Heap[i]$ is the node in $Heap[\lfloor i/2 \rfloor]$. A pointer *Tail* points at the first free location in the heap (the leftmost vacant location in the last level of the heap). *Tail* is initialized to 1.

Insert adds a node to the heap by putting it in the location pointed at by *Tail*. *Tail* is then incremented by 1. Finally, the node is floated up along the path from its entry location towards the root, in each step swapped with its parent, until it reaches a location where its parent's value is less than its own value. if *Tail*$= N$ when *Insert* is called, *Insert* returns *FAILURE*. *DeleteMin* removes the root of the heap (which has the least value of all the nodes in the heap) and returns it. Then the rightmost node in the last level of the heap is moved to the root and *Tail* is decremented by 1. Finally, this node is seeped down by repeatedly swapping it with its least son, until it reaches a location where both its sons are greater than it. if when *DeleteMin* is called *Tail*$= 1$, *DeleteMin* returns *FAILURE*. The *Insert* and *DeleteMin* procedures described above ensure that the heap is always kept as a complete binary tree with depth of $O(\log n)$, where n is the number of nodes in the heap.

3.1 The Data Structures And The Routines

We use the following definitions: An ***ascending node*** is a node that is in the middle of being inserted: a location was seized for the node, but the node has not yet reached its correct location. The ***owner*** of an ascending node is the process that initiated the *Insert* operation of that node. A ***descending node*** is a node that has replaced a deleted root and it is in the middle of being seeped down from the root, but has not yet reached its correct location. An ***independent node*** is a node that is neither descending nor ascending.

We augment the data structure of the sequential implementation. A node is represented as a triplet (*value*, *type*, *freeze*). *value* is an integer which specifies the node's priority (lower values correspond to nodes with higher priorities). *value* can be assigned two extra values, $-\infty$ and ∞, where ∞ denotes an empty node and $-\infty$ denotes a deleted root. *type* is *UP* for an ascending node, *DOWN* for a descending node, and *IND* otherwise. *freeze* is a binary field that implements the ***freezing token***, whose role will be explained in the next subsection. A node in the priority-queue can be in one of the following forms (where '–' stands for either *TRUE* or *FALSE*): (∞,*IND*,-) – an empty node, (*val*,*UP*,–)

– an ascending node, *(val,DOWN,-)* – a descending node, *(val,IND,-)* – an independent node, *(−∞,IND,-)* – a root that has been deleted. *Heap*[*i*] for *i* in [1..*N*] is initiated to *(∞,IND,FALSE)*. For convenience we hold an extra location, *Heap*[0], which is initiated to *(−∞,IND,-)*. *Tail* is a pair (*pointer*, *freeze*), which is initialized to (1,*FALSE*). Roughly speaking, nodes are inserted to the location pointed at by *Tail.pointer*.

The *Insert* operation (Figure 1) is implemented in two stages. First a location is seized for the new node, using the *SeizeTail* procedure (Figure 1). Then, the node is floated up according to its priority, by calling the *FloatUp* procedure (Figure 2). The *DeleteMin* operation (Figure 3) is implemented by calling the *DeleteRoot* procedure (Figure 3), which first gets an independent node to the root (if needed) and then deletes that node and returns it. The above routines use the following subroutines: *GetNonAsc* (Figure 2) – gets to a specified location a non-ascending node. *GetNonDes* (Figure 4) – gets to a specified location a non-descending node. *SwapRoot* – finds a replacement node for a deleted root and replaces the deleted root with that replacement node. The algorithm for *SwapRoot* can be found in [11].

3.2 Manipulating *Tail* — *SeizeTail* and *SwapRoot*

In this subsection we describe the parts of the algorithm that manipulate *Tail*. These are *SeizeTail* and *SwapRoot*.

SeizeTail (Figure 1) works as follows: if *Tail* points at an empty location, *SeizeTail* tries to swap that empty location with the new node, using *SC* – this seizes the location. If however the location pointed at by *Tail* is not empty then there are two possibilities: If *Tail*= *N* *SeizeTail* returns *FAILURE*, otherwise, *Tail* is incremented and the whole process is repeated.

When a process that executes *DeleteMin* finds out that the root is already deleted, it calls *SwapRoot*. *SwapRoot* works as follows: If *Tail*= 1 the deleted root is swapped by an empty node and *SwapRoot* returns. Otherwise, a replacement for the deleted root must be found. As will be explained later on, the replacement must be a non-ascending node. If *Tail* points at an empty location *Tail* must be decremented. When *Tail* points at a non-empty location a non-ascending node is brought to the location pointed at by *Tail* (using *GetNonAsc*, which will be described later on). Finally, the node is moved to the root, marked as descending, and the location pointed at by *Tail* is made an empty location, by a single *SC2* instruction.

There are two problems that have to be dealt with when manipulating *Tail*:

1. *Tail* must be incremented and decremented using a protocol that ensures that no gaps will occur and that non-empty nodes will not be stepped over.
2. *Tail* may suffer from a *ping-pong effect*: A process that tries to insert a node may increment *Tail* to point at an empty location, and just before it puts the new node in the empty location, another process, that tries to find a node to replace a deleted root, might decrement *Tail* to point at a non-empty location, and so on.

```
Insert(val)

t := SeizeTail(val);
if (t = FAILURE) then return (FAILURE);
FloatUp(t, val);
return (SUCCESS);



SeizeTail(val)

while (TRUE) do
    t := LL(Tail);
    cur := LL(Heap[t.ptr]);
    if ( Empty(cur) and not Frozen(cur) ) then
        if ( SC(Heap[t.ptr], (val, UP, FALSE)) ) then
            return (t.ptr);
    else if ( Empty(cur) and Frozen(cur) ) then
        SC&V(Heap[t.ptr], (∞, IND, FALSE), Tail);
    else if (t.ptr = N) then
        return (FAILURE);
    else if ( (not Deleted(cur)) and (not Frozen(t))
            and (not Frozen(cur)) ) then
        SC&V(Tail, (t.ptr + 1, t.frz), Heap[t.ptr]);
    else    /* Frozen(t) or Deleted(cur) or */
            /* (Frozen(cur) and (not Empty(cur)) */
        SwapRoot();
end while
```

Fig. 1. The non-blocking algorithms for *Insert* and *SeizeTail*

In order to solve the first problem we use the following rules: *Tail* may be increment only if it points at a non-empty location. *Tail* may be decrement only if it points at an empty location. A new node can be put only at an empty location.

The second problem is solved as follows: If there exists an operation on the data structure that takes more than one atomic instruction, and must not be interrupted until completed, the operation is *frozen* using a *freezing token*. The operation is divided into steps, such that each step can be completed using a single atomic instruction. The process that initiates the operation puts a token on the memory location which is to be accessed by the first step. As the operation progresses, the token is moved to the memory location that is to be accessed by the next step. Writing to a memory location, removing the token from that location and moving the token to next memory location, are all executed by a single *SC2* instruction. When a process finds a memory location with its freezing token on, it must complete the operation before it can write to this memory location. The location of the freezing token enables the process to know what is the current step of the operation. Writing to a memory location may also be conditioned on another location not being frozen, by using an atomic instruction that accesses both locations. In the last step, the token is removed. This technique enables a mode of operation which resembles locking in a non-blocking implementation.

We now describe how the freezing token technique is used to prevent the ping-pong effect. When the root is deleted it is frozen, by setting its *freeze* field to *TRUE*. *SwapRoot* first freezes *Tail* by setting *Heap*[1].*freeze* to *FALSE* and *Tail.freeze* to *TRUE* by a single *SC2* instruction (this moves the freezing token from *Heap*[1] to *Tail*). If the frozen *Tail* points at an empty location, *Tail* must be decremented. When *Tail* is decremented, the empty location pointed at by *Tail* must be frozen, to prevent occupying this location. For this reason we use *SC2* to decrement *Tail*. When the frozen *Tail* points at a non-empty location, the freezing token is moved from *Tail* to the non-empty location. Then a non-ascending node is brought to the frozen location (using *GetNonAsc*, which will be described later on). Finally, the node is moved to the root, marked as a descending node, its *freeze* field is set to off and the location pointed at by *Tail* is made an empty location, by a single *SC2* instruction. If a process that executes *SeizeTail* observes a frozen *Tail* that points at a non-empty location, or a *Tail* that points at a frozen non-empty location, the process cannot increment *Tail*. Since the process also cannot wait for *Tail* or for the location to be defrosted, it must call *SwapRoot* to complete the operation.

Under the assumption that *GetNonAsc* is non-blocking, it can be shown that *SeizeTail* and *SwapRoot* are non-blocking. A failure to complete *SeizeTail* or *SwapRoot* in a certain iteration by some process, must be the result of another process (executing either *SeizeTail* or *SwapRoot*) success. However, *SeizeTail* and *SwapRoot* are not wait-free, since a process might suffer starvation: A process that executes *SeizeTail* may fail for ever because each time it tries, another process may be ahead of it. The same is true for *SwapRoot*.

3.3 The *FloatUp* and *GetNonAsc* Procedures

When a location is seized for a node by *SeizeTail*, the node is marked as ascending. Then, the node is floated up towards the root by the *FloatUp* procedure (Figure 2). Floating an ascending node resembles bubble sort -- in each step *FloatUp* calls *GetNonAsc* (Figure 2), where the ascending node is compared with its parent, and the two of them are swapped if the ascending node's value is less than its parent's value. Swapping the nodes is executed using the *SC2* primitive, which ensures that the swapped nodes are really those meant to be swapped.

```
FloatUp(t, val)

while (TRUE) do
    cur := LL(Heap[t]);
    if ( (cur.val = val) and Asc(cur) ) then
        t := GetNonAsc(t,cur);
    else if ( (cur.val ≤ val) and (cur.type = IND) ) then
        return (SUCCESS);
    else
        t := parent(t);
end while


GetNonAsc(t,cur)

while ( Asc(cur) ) do
    par := LL(Heap[parent(t)]);

    if (par.type = IND) then
        if (par.val ≤ cur.val) then /* make cur independent */
            if ( SC(Heap[t], (cur.val,IND,cur.frz) ) then
                break ;
        else /* swap cur and par */
            if ( SC2(Heap[t], (par.val, IND, cur.frz),
                    Heap[parent(t)], (cur.val, UP, par.frz) ) then
                t := parent(t);     break ;

    else if Des(par) then
        if (GetNonDes(parent(t),par) = t) then
            t := parent(t);     break ;

    else /* Asc(par) */
        GetNonAsc(parent(t),par);
    cur := LL(Heap[t]);

end while

return (t);
```

Fig. 2. The non-blocking algorithms for *FloatUp* and *GetNonAsc*

The owner of node v, P_i, floats v up until v's parent is an independent node with value less than v's value. P_i then makes v independent and returns. An independent node satisfies the property that all the non-descending nodes on the path from it to the root have values less than or equal to its own value.

If P_i observes that v's parent, u, is a descending node, P_i calls *GetNonDes* (Figure 4) which either seeps u one location down, or makes u independent. *GetNonDes* is described in the next subsection. If P_i observes that v's parent, u, is an ascending node, it cannot make v independent, even if u's value is less than v's value, since there may still be nodes in the path from u to the root with values greater than v's value. Neither can P_i swap v and u, since this would cause u to move down and as will be understood from the next paragraph, an ascending node must not move down. Moreover, P_i cannot wait for u to be floated up by u's owner. Therefore, P_i floats u one location up, or makes u independent, by calling *GetNonAsc* recursively.

Since one process may float a node owned by another process, a process may lose its node. The owner P_i of node v must not return before v is made independent and the *Insert*(v) operation is completed. Therefore, if P_i loses v, P_i locates v by scanning the path from the last location it observed v, towards the root, until it reaches the first independent node with value less than or equal to v's value. Since an ascending node can only move upwards, if v is not located, it must have been made independent and even might have been deleted from the priority-queue. If v is not located or if it is found to be independent, *FloatUp* returns. This also explains why an ascending node cannot be used as a replacement for a deleted root: locating it would cost $O(n)$ time.

Under the assumption that *GetNonDes* is non-blocking, it can be shown that *FloatUp* and *GetNonAsc* are non-blocking as well. However, *FloatUp* and *GetNonAsc* are not wait-free, since an ascending node's parent may change again and again as an infinite number of ascending and descending nodes move by the node (and as was explained, an ascending node can be made independent only if its parent is an independent node, whose value is less than the ascending node's value).

3.4 *DeleteRoot* and *GetNonDes*

A process that executes *DeleteRoot* (Figure 3) acts according to the type of the root. An independent node: deletes the root and returns it. An ascending node: calls *GetNonAsc*(1). A descending node: calls *GetNonDes*(1). A deleted node: calls *SwapRoot* to find a replacement for the root (which will be seeped down later on). An empty node: returns *FAILURE*.

GetNonDes (Figure 4) gets to a specified location a non-ascending node. First, *GetNonDes* makes sure that both sons of the specified location are non-descending (by calling *GetNonDes* recursively for each of the sons, if needed). Then, *GetNonDes* repeatedly tries to either make the node in the specified locations to be independent (if it is less than both its sons), or to swap the node in the specified location with its least son (otherwise).

```
DeleteMin()
return (DeleteRoot());


DeleteRoot()

while (TRUE) do
    root := LL(Heap[1]);
    case root of
        Ind:      if ( SC(Heap[1], (-∞,IND,TRUE)) ) then return (root.val);
        Empty:    return (FAILURE);
        Asc:      SC(Heap[1], (root.val, IND, root.frz));
        Des:      GetNonDes(1,root);
        Deleted:  SwapRoot();
    end case
end while
```

Fig. 3. The non-blocking algorithm for *DeleteMin*

3.5 Correctness Proof

We define the occurrence time of an *Insert*(v) operation as the time v was made independent. We define the occurrence time of a *DeleteMin* operation as the time the root is deleted for that operation. The correctness is implied by the following lemmas:

Lemma 1. *Locations are seized in order - with no gaps and without stepping over non-empty locations.*

Lemma 2. SeizeTail*(v) returns a value t other than* FAILURE *iff the location* Heap[t] *is seized for v. If* SeizeTail*(v) returns* FAILURE *then there exists a time within the execution interval of* SeizeTail *in which the heap is full.*

Lemma 3. *An ascending node can only move up and a descending node can only move down.*

Corollary: An ascending node cannot be passed by another ascending node and a descending node cannot be passed by another descending node.

Lemma 4. *Any non-ascending node v satisfies that each of the independent nodes on the path from v to the root has a value that is less than or equal to v's value and each of the ascending nodes on the path from v to the root has a value that is strictly less than v's value.*

```
GetNonDes(t, cur)

l := left(t);
r := right(t);

while (Des(cur)) do

    /* Call GetNonDes recursively to make sure both sons are non-descending. */
    lson := LL(Heap[l]);
    if Des(lson) then GetNonDes(l, lson);
    rson := LL(Heap[r]);
    if Des(rson) then GetNonDes(r, rson);

    while ( VL(Heap[t]) ) do

        lson := LL(Heap[l]);
        rson := LL(Heap[r]);
        if ( (lson.val < cur.val) or (rson.val < cur.val) ) then
        /* One of cur's sons is less than cur – swap cur with its least son. */
            if ( (lson.val < rson.val) or ((lson.val = rson.val) and Ind(lson)) ) then
                if ( SC2(Heap[t], (lson.val, lson.type, cur.frz),
                             Heap[l], (cur.val, cur.type, lson.frz)) ) then
                    return (l);
            else
                if ( SC2(Heap[t], (rson.val, rson.type, cur.frz),
                             Heap[r], (cur.val, cur.type, rson.frz)) ) then
                    return (r);

        else
        /* cur is less than both its sons – make cur independent. */
            if ( SC(Heap[t],(cur.val,IND,cur.frz)) ) then
                return (t);

    end while
    cur := LL(Heap[t]);
end while
return (t);
```

Fig. 4. The non-blocking algorithm for *GetNonDes*

Lemma 5. *Let v be an ascending node last observed by process P_i in location x. If P_i fails to locate a node with the same value as v by scanning the path from x to the root, then v must have already been made independent.*

Lemma 6. *Let P_i be the owner of node v. If P_i returns* SUCCESS *from* Insert(v), *v has already been made independent. If P_i returns* FAILURE *from* Insert(v), *then a legal occurrence time (within the execution interval of the* Insert *operation) in which the heap was full, can be defined.*

Corollary: The occurrence time of inserting a node v, can be defined as the time v was made independent.

Lemma 7. *Let v be a node returned by process P_i that executes* DeleteMin. *v is then the node with the least value of all nodes in the priority-queue that were inserted before v was deleted (and that have not been deleted from the priority-queue before v was deleted). If however P_i returns* FAILURE *from* DeleteMin, *then a legal occurrence time (within the execution interval of the* DeleteMin *operation) in which the heap was empty, can be defined.*

Lemma 8. *The algorithm is non-blocking: Under the assumption that at any given time, eventually some process executes an instructions, at any given time, eventually an operation initiated by some process is completed.*

3.6 Time And Space Complexity Analysis

In this section we briefly sketch the time complexity analysis for executing a set of r *Insert* and *DeleteMin* operations. Let n be the maximum number of nodes in the heap during the execution of the r operations. We define a *step* of node v as the event of v moving one location (from parent to son or from son to parent). Since an ascending node can only move up, it can step at most $\log n$ steps. In the same way, a descending node can step at most $\log n$ steps as well. The time complexity is computed using the following lemma:

Lemma 9. *Any iteration consisting of $O(1)$ instructions, which is executed in any of the subroutines, can be credited to one of the following events, such that at most $O(1)$ iterations, executed by a specific process, are credited to the same event: A step of a non-independent node, seizing a location for a new node, deleting the root and making a node to be independent.*

In a set of r operations there are at most $O(r \log n)$ steps of non-independent nodes and at most $O(r)$ events of seizing locations for new nodes, deleting the root and making nodes to be independent. Together we get a total of at most $O(r \log n)$ events. Since in a set of r operations there are at most $O(r \log n)$ events, each one of them is credited for at most $O(1)$ iterations consisting of $O(1)$ instructions, executed by a specific process, then each process can execute at most $O(r \log n)$ instructions. Therefore all processes together execute at most $O(pr \log n)$ instructions during the set of r operations.

4 The Wait-Free Implementation

The non-blocking implementation can be made wait-free, by a technique presented in this section. The technique is inspired by the *operation combining* technique [8]. However, operation combining suffers from all the disadvantages described earlier, since each process operates on a local copy of the shared object.

4.1 Making the Non-Blocking Implementation Wait-Free

We hold a shared array $Req[1..p]$, where p is the number of processes, called the *shared request array*. We also hold, for each process P_i, an array $LocalReq_i[1..p]$, called P_i's *local request array*. All the entries in the shared request array are initialized to a state that denotes that there is no pending request.

Before executing an operation (e.g. seizing a location for a new node), P_i issues a request for that operation in $Req[i]$. P_i then copies $Req[1..p]$ to $LocalReq_i$, and tries to execute each one of the requests registered in $LocalReq_i$, until all of them are fulfilled (either by P_i, or by some other process). Requests may be fulfilled not in the order in which they were issued; the correctness, however, is not violated, since each request is fulfilled within the execution interval of the corresponding operation.

When P_i fulfills a request issued by process P_j, P_i marks the request as fulfilled in $Req[j]$. This enables the other processes (and P_j in particular) to learn that the request had been fulfilled and that they can move on. P_i must not mark the request as fulfilled *before* it had fulfilled the request, because it might fail executing the request or it might even halt. P_i must also not mark the request as fulfilled *after* it had fulfilled the request, since other processes might try to fulfill the request before P_i marks it as fulfilled, and then the request might be fulfilled more than once (e.g. more than one location seized for the same new node). Therefore, both executing a request and marking the request as fulfilled in $Req[j]$ must be done simultaneously, by a single atomic instruction. In case fulfilling a request in the non-blocking algorithm uses a single *SC* instruction, this is performed by a single *SC2* instruction in the wait-free algorithm which replaces the *SC* instruction the non-blocking algorithms.

Measures must be taken to ensure that after a request is fulfilled, a process that tries to fulfill that request continues to execute only a bounded number of instructions before it learns that the request had been fulfilled (and returns). One way to do this is to augment all the loop-conditions to check that the request is not fulfilled yet.

4.2 Proving that the Technique Yields a Wait-Free Implementation

We now show that this technique yields a wait-free implementation. Let t_1 be the time in which P_i finishes copying $Req[1..p]$ and let t_2 be the time in which all the requests registered in $LocalReq_i$ are fulfilled. The number of requests fulfilled by all processes together within the interval $[t_1, t_2]$ is bounded by $2p - 1$:

- At most p unfulfilled requests that were issued before t_1 (registered in $Req[1..p]$ in t_1).
- At most $p-1$ requests that are issued within $[t_1, t_2]$: All the requests registered in $LocalReq_i$ that are not fulfilled, are still registered in $Req[1..p]$. A process P_j that issues a request within $[t_1, t_2]$ observes these pending requests in Req, copies them to $LocalReq_j$ and does not issue another request before these requests are fulfilled.

It can be proved that the algorithm to fulfill a request (e.g. *SeizeTail*) is still non-blocking. Since the number of the requests that can be executed within $[t_1, t_2]$ is bounded, since the algorithm to fulfill a request is non-blocking, and since after a request is fulfilled a process that tries to fulfill that request continues to executes a bounded number of instructions before it learns that the request had been fulfilled, we get a wait-free implementation. It can be proved that the correctness for is not violated. The wait-free algorithms are described with detail in [11].

4.3 Complexity Analysis and Performance

The time complexity for executing a set of r *Insert* and *DeleteMin* operations in the wait-free implementation is $O(rp(p+\log n))$, which is the sum of $O(rp \log n)$ (the corresponding time complexity for the non-blocking implementation) and $O(rp^2)$ (the extra work for scanning the requests arrays). This time complexity is compared with $O(rp(p+n))$ in [8]. The space complexity is $O(n+p^2)$, compared with $O((n+p)p)$ in [8].

Making the implementation wait-free degrades overall system performance. Therefore, if the wait-free property is not required, the non-blocking implementation should be generally preferred over a wait-free implementation. If the wait-free property is required, the level of wait-freedom can be controlled, by having a process copy the requests array and trying to fulfill the requests registered there only after some constant number, k, of requests it had fulfilled for itself. With $k = 1$ we get the current wait-free implementation. With $k = \infty$ we get the current non-blocking implementation.

5 Conclusions

The primary problems that have to be dealt with in an effectively parallel wait-free implementation of a concurrent object are:

- An operation on the object may be started before previous operations are completed, so the object may be in an inconsistent state while more than one operation is in progresses.
- There may exist parts of an operation on the object that take more than one atomic instruction and must not be interrupted until completed.

We have presented three general techniques that may be used in converting a sequential object to a wait-free, effectively parallel, concurrent object:

- Marking words in memory as consistent (independent) or inconsistent (non-independent).
- The use of a freezing token.
- A technique for converting a non-blocking implementation to a wait-free one.

Using these techniques, we have presented a time and space efficient, effectively parallel, wait-free implementation of a concurrent priority-queue, based on a heap data structure. It would be interesting to find a lower bound of wait-free implementations of a concurrent priority-queue, given the set of allowed atomic primitives.

References

1. A. V. Aho, J.E. Hopcroft, J. D. Ullman. Data Structures and Algorithms. pages 139–145.
2. J. Alemany, E. W. Felten. Performance Issues in Non-blocking Synchronization on Shared-memory Multiprocessors. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 124–134, August 1992.
3. Aspnes J. and M. Herlihy, Fast Randomized Consensus Using Shared Memory, *Jour. of Algorithms*, Vol. 11, pages 441–461, September 1990.
4. J. Aspnes, Time– and Space–Efficient Randomized Consensus, *Proceedings of the 9th ACM Conference on Principles of Distributed Computing,* August 1990, pages 325–331.
5. R. J. Anderson, H. Woll. Wait-Free parallel algorithms for the union-find problem. In *Proceedings of the 23rd ACM Symposium on Theory of Computation*, pages 370–380, May 1991.
6. B. Chor, A. Israeli, and M. Li, "On Processors Coordination Using Asynchronous Hardware, *Proceedings of the 6th ACM Conference on Principles of Distributed Computing*, pages 86–97, August 1987.
7. M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1988.
8. M. P. Herlihy. A methodology for implementing highly concurrent data structures. DEC Cambridge Research Lab Technical report 91/10.
9. M. P. Herlihy, J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. DEC Cambridge Research Lab Technical report 92/7.
10. M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. In *ACM TOPLAS*, 12(3):463-492, 1990.
11. A. Israeli and L. Rappoport, Efficient Wait-Free Implementation of a Concurrent Priority Queue. Tehnion, Faculty of Computer Science, Technical report 781.
12. Loui M. C. and H. H. Abu-Amara, Memory Requirements for Agreement among Unreliable Asynchronous Processes, *Advances in Computing Research,* JAI press, 1987, pages 163–183.
13. L. Lamport, "On Interprocess Communication. Part I: Basic Formalism", Distributed Computing 1, 2 1986, pages 77–85.
14. L. Lamport, "On Interprocess Communication. Part II: Algorithms", Distributed Computing 1, 2 1986, pages 86–101.
15. G.L. Peterson, Concurrent reading while writing, *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 1, pages 46–55.