Technical Report no. 2003-01

Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems[1]

Håkan Sundell **Philippas Tsigas**

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2003

[1] This work is partially funded by: i) the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se) supported by the Swedish Foundation for Strategic Research and ii) the Swedish Research Council for Engineering Sciences.

Technical Report in Computing Science at
Chalmers University of Technology and Göteborg University

Technical Report no. 2003-01
ISSN: 1650-3023

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2003

Abstract

We present an efficient and practical lock-free implementation of a concurrent priority queue that is suitable for both fully concurrent (large multi-processor) systems as well as pre-emptive (multi-process) systems. Many algorithms for concurrent priority queues are based on mutual exclusion. However, mutual exclusion causes blocking which has several drawbacks and degrades the system's overall performance. Non-blocking algorithms avoid blocking, and are either lock-free or wait-free. Previously known non-blocking algorithms of priority queues did not perform well in practice because of their complexity, and they are often based on non-available atomic synchronization primitives. Our algorithm is based on the randomized sequential list structure called Skiplist, and a real-time extension of our algorithm is also described. In our performance evaluation we compare our algorithm with some of the most efficient implementations of priority queues known. The experimental results clearly show that our lock-free implementation outperforms the other lock-based implementations in all cases for 3 threads and more, both on fully concurrent as well as on pre-emptive systems.

1 Introduction

Priority queues are fundamental data structures. From the operating system level to the user application level, they are frequently used as basic components. For example, the ready-queue that is used in the scheduling of tasks in many real-time systems, can usually be implemented using a concurrent priority queue. Consequently, the design of efficient implementations of priority queues is a research area that has been extensively researched. A priority queue supports two operations, the $Insert$ and the $DeleteMin$ operation. The abstract definition of a priority queue is a set of key-value pairs, where the key represents a priority. The $Insert$ operation inserts a new key-value pair into the set, and the $DeleteMin$ operation removes and returns the value of the key-value pair with the lowest key (i.e. highest priority) that was in the set.

To ensure consistency of a shared data object in a concurrent environment, the most common method is to use mutual exclusion, i.e. some form of locking. Mutual exclusion degrades the system's overall performance [14] as it causes blocking, i.e. other concurrent operations can not make any progress while the access to the shared resource is blocked by the lock. Using mutual exclusion can also cause deadlocks, priority inversion (which can be solved efficiently on uni-processors [13] with the cost of more difficult analysis, although not as efficient on multiprocessor systems [12]) and even starvation.

To address these problems, researchers have proposed non-blocking algorithms for shared data objects. Non-blocking methods do not involve mutual exclusion, and therefore do not suffer from the problems that blocking can cause. Non-blocking algorithms are either lock-free or wait-free. Lock-free implementations guarantee that regardless of the contention caused by concurrent operations and the interleaving of their sub-operations, always at least one operation will progress. However, there is a risk for starvation as the progress of other operations could cause one specific operation to never finish. This is although different from the type of starvation that could be caused by blocking, where a single operation could block every other operation forever, and cause starvation of the whole system. Wait-free [6] algorithms are lock-free and moreover they avoid starvation as well, in a wait-free algorithm every operation is guaranteed to finish in a limited number of steps, regardless of the actions of the concurrent operations. Non-blocking algorithms have been shown to be of big practical importance in practical applications [17, 18], and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [16].

There exist several algorithms and implementations of concurrent priority queues. The majority of the algorithms are lock-based, either with a single lock on top of a sequential algorithm, or specially constructed algorithms using multiple locks, where each lock protects a small part of the shared data structure. Several different representations of the shared data structure are used, for example: Hunt *et al.* [7] presents an implementation which is based on heap structures, Grammatikakis *et al.* [3] compares different structures including cyclic arrays and heaps, and most recently Lotan and Shavit [9] presented an implementation based on the Skiplist structure [11]. The algorithm by Hunt *et al.* locks each node separately and uses a technique to scatter the accesses to the heap, thus reducing the contention. Its implementation is publicly available and its performance has been documented on multi-processor systems. Lotan and Shavit extend the functionality of the concurrent priority queue and assume the availability of a global high-accuracy clock. They apply a lock on each pointer, and as the multi-pointer based Skiplist structure is used, the number of locks is significantly more than the number of nodes. Its performance has previously only been documented by simulation, with very promising results.

Israeli and Rappoport have presented a wait-free algorithm for a concurrent priority queue [8]. This algorithm makes use of strong atomic synchronization primitives that have not been implemented in any currently existing platform. However, there exists an attempt for a wait-free algorithm by Barnes [2] that uses existing atomic primitives, though this algorithm does not comply with the generally accepted definition of the wait-free property. The algorithm is not yet implemented and the theoretical analysis predicts worse behavior than the corresponding sequential algorithm, which makes it not of practical interest.

One common problem with many algorithms for concurrent priority queues is the lack of precise defined semantics of the operations. It is also seldom that the correctness with respect to concurrency is proved, using a strong property like linearizability [5].

In this paper we present a lock-free algorithm of a concurrent priority queue that is designed for efficient use in both pre-emptive as well as in fully concurrent environments. Inspired by Lotan and Shavit [9], the algorithm is based on the randomized Skiplist [11] data structure, but in contrast to [9] it is lock-free. It is also implemented using common synchronization primitives that are available in modern systems. The algorithm is described in detail later in this paper, and the aspects concerning the underlying lock-free memory management are also presented. The precise semantics of the operations are defined and a proof is given that our implementation is lock-free and linearizable. We have performed experiments that compare the performance of our algorithm with some of the most efficient implementations of concurrent priority queues known, i.e. the implementation by Lotan and Shavit [9] and the implementation by Hunt *et al.* [7]. Experiments were performed on three different platforms, consisting of a multiprocessor system using different operating systems and equipped with either 2, 4 or 64 processors. Our results show that our algorithm outperforms the other lock-based implementations for 3 threads and more, in both highly pre-emptive as well as in fully concurrent environments. We also present an extended version of our algorithm that also addresses certain real-time aspects of the priority queue as introduced by Lotan and Shavit [9].

The rest of the paper is organized as follows. In Section 2 we define the properties of the systems that our implementation is aimed for. The actual algorithm is described in Section 3. In Section 4 we define the precise semantics for the operations on our implementations, as well showing correctness by proving the lock-free and linearizability property. The experimental evaluation that shows the performance of our implementation is presented in Section 5. In Section 6 we extend our algorithm with functionality that can be needed for specific real-time applications. We conclude the paper with Section 7.

2 System Description

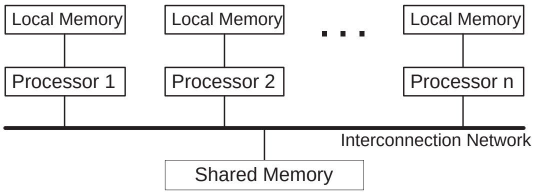

Figure 1. Shared Memory Multiprocessor System Structure

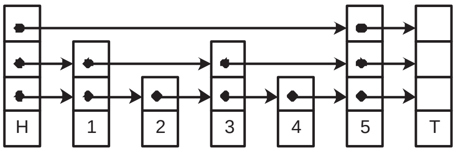

Figure 2. The Skiplist data structure with 5 nodes inserted.

A typical abstraction of a shared memory multi-processor system configuration is depicted in Figure 1. Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co-operating tasks is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can serve (run) many tasks at a time. The co-operating tasks, possibly running on different processors, use shared data objects built in the shared memory to co-ordinate and communicate. Tasks synchronize their operations on the shared data objects through sub-operations on top of a cache-coherent shared memory. The shared memory may not though be uniformly accessible for all nodes in the system; processors can have different access times on different parts of the memory.

3 Algorithm

The algorithm is based on the sequential Skiplist data structure invented by Pugh [11]. This structure uses randomization and has a probabilistic time complexity of $O(logN)$ where N is the maximum number of elements in the list. The data structure is basically an ordered list with randomly distributed short-cuts in order to improve search times, see Figure 2. The maximum height (i.e. the maxi-

```
structure Node
  key,level,validLevel ⟨,timeInsert⟩ : integer
  value : pointer to word
  next[level],prev : pointer to Node
```

Figure 3. The Node structure.

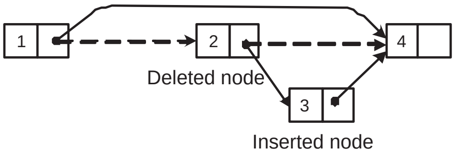

Figure 4. Concurrent insert and delete operation can delete both nodes.

```
function TAS(value:pointer to word):boolean
    atomic do
        if *value=0 then
            *value:=1;
            return true;
        else return false;

procedure FAA(address:pointer to word, number:integer)
    atomic do
        *address := *address + number;

function CAS(address:pointer to word, oldvalue:word,
 newvalue:word):boolean
    atomic do
        if *address = oldvalue then
            *address := newvalue;
            return true;
        else return false;
```

Figure 5. The Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives.

mum number of next pointers) of the data structure is $logN$. The height of each inserted node is randomized geometrically in the way that 50% of the nodes should have height 1, 25% of the nodes should have height 2 and so on. To use the data structure as a priority queue, the nodes are ordered in respect of priority (which has to be unique for each node), the nodes with highest priority are located first in the list. The fields of each node item are described in Figure 3 as it is used in this implementation. For all code examples in this paper, code that is between the “⟨” and “⟩” symbols are only used for the special implementation that involves timestamps (see Section 6), and are thus not included in the standard version of the implementation.

In order to make the Skiplist construction concurrent and non-blocking, we are using three of the standard atomic synchronization primitives, Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS). Figure 5 describes the specification of these primitives which are available in most modern platforms.

As we are concurrently (with possible preemptions) traversing nodes that will be continuously allocated and reclaimed, we have to consider several aspects of memory management. No node should be reclaimed and then later re-allocated while some other process is traversing this node. This can be solved for example by careful reference counting. We have selected to use the lock-free memory management scheme invented by Valois [19] and corrected by Michael and Scott [10], which makes use of the FAA and CAS atomic synchronization primitives.

To insert or delete a node from the list we have to change the respective set of next pointers. These have to be changed consistently, but not necessary all at once. Our solution is to have additional information on each node about its deletion (or insertion) status. This additional information will guide the concurrent processes that might traverse into one partial deleted or inserted node. When we have changed all necessary next pointers, the node is fully deleted or inserted.

One problem, that is general for non-blocking implementations that are based on the linked-list structure, arises when inserting a new node into the list. Because of the linked-list structure one has to make sure that the previous node is not about to be deleted. If we are changing the next pointer of this previous node atomically with CAS, to point to the new node, and then immediately afterwards the previous node is deleted - then the new node will be deleted as well, as illustrated in Figure 4. There are several solutions to this problem. One solution is to use the CAS2 operation as it can change two pointers atomically, but this operation is not available in any existing multiprocessor system. A second solution is to insert auxiliary nodes [19] between each two normal nodes, and the latest method introduced by Harris [4] is to use one bit of the pointer values as a deletion mark. On most modern 32-bit systems, 32-bit values can only be located at addresses that are evenly dividable by 4, therefore bits 0 and 1 of the address are always set to zero. The method is then to use the previously unused bit 0 of the next pointer to mark that this node is about to be deleted, using CAS. Any concurrent $Insert$ operation will then be notified about the deletion, when its CAS operation will fail.

One memory management issue is how to de-reference pointers safely. If we simply de-reference the pointer, it might be that the corresponding node has been reclaimed before we could access it. It can also be that bit 0 of the pointer was set, thus marking that the node is deleted, and therefore the pointer is not valid. The following functions are defined for safe handling of the memory management:

function READ_NODE(address:**pointer to pointer to** Node):**pointer to** Node /* De-reference the pointer and increase the reference counter for the corresponding node. In case the pointer is marked, NULL is returned */

```
// Global variables
head,tail : pointer to Node
// Local variables
node2 : pointer to Node

function ReadNext(node1:pointer to pointer to Node,
 level:integer):pointer to Node
if IS_MARKED((*node1).value) then
      *node1:=HelpDelete(*node1,level);
node2:=READ_NODE((*node1).next[level]);
while node2=NULL do
      *node1:=HelpDelete(*node1,level);
      node2:=READ_NODE((*node1).next[level]);
return node2;

function ScanKey(node1:pointer to pointer to Node,
 level:integer, key:integer):pointer to Node
node2:=ReadNext(node1,level);
while node2.key < key do
      RELEASE_NODE(*node1);
      *node1:=node2;
      node2:=ReadNext(node1,level);
return node2;
```

Figure 6. Functions for traversing the nodes in the Skiplist data structure.

function COPY_NODE(node:**pointer to** Node):**pointer to** Node /* Increase the reference counter for the corresponding given node */

procedure RELEASE_NODE(node:**pointer to** Node) /* Decrement the reference counter on the corresponding given node. If the reference count reaches zero, then call RELEASE_NODE on the nodes that this node has owned pointers to, then reclaim the node */

While traversing the nodes, processes will eventually reach nodes that are marked to be deleted. As the process that invoked the corresponding $Delete$ operation might be pre-empted, this $Delete$ operation has to be helped to finish before the traversing process can continue. However, it is only necessary to help the part of the $Delete$ operation on the current level in order to be able to traverse to the next node. The function $ReadNext$, see Figure 6, traverses to the next node of *node1* on the given level while helping (and then sets *node1* to the previous node of the helped one) any marked nodes in between to finish the deletion. The function $ScanKey$, see Figure 6, traverses in several steps through the next pointers (starting from *node1*) at the current level until it finds a node that has the same or higher key (priority) value than the given key. It also sets *node1* to be the previous node of the returned node.

The implementation of the $Insert$ operation, see Fig-

```
// Local variables
node1,node2,newNode,
 savedNodes[maxlevel]: pointer to Node

function Insert(key:integer, value:pointer to word):boolean
⟨TraverseTimeStamps();⟩
level:=randomLevel();
newNode:=CreateNode(level,key,value);
COPY_NODE(newNode);
node1:=COPY_NODE(head);
for i:=maxLevel-1 to 1 step -1 do
      node2:=ScanKey(&node1,i,key);
      RELEASE_NODE(node2);
      if i<level then savedNodes[i]:=COPY_NODE(node1);
while true do
      node2:=ScanKey(&node1,0,key);
      value2:=node2.value;
      if not IS_MARKED(value2) and node2.key=key then
         if CAS(&node2.value,value2,value) then
            RELEASE_NODE(node1);
            RELEASE_NODE(node2);
            for i:=1 to level-1 do
               RELEASE_NODE(savedNodes[i]);
            RELEASE_NODE(newNode);
            RELEASE_NODE(newNode);
            return $true_2$;
         else
            RELEASE_NODE(node2);
            continue;
      newNode.next[0]:=node2;
      RELEASE_NODE(node2);
      if CAS(&node1.next[0],node2,newNode) then
         RELEASE_NODE(node1);
         break;
      Back-Off
for i:=1 to level-1 do
      newNode.validLevel:=i;
      node1:=savedNodes[i];
      while true do
         node2:=ScanKey(&node1,i,key);
         newNode.next[i]:=node2;
         RELEASE_NODE(node2);
         if IS_MARKED(newNode.value) or
          CAS(&node1.next[i],node2,newNode) then
            RELEASE_NODE(node1);
            break;
         Back-Off
newNode.validLevel:=level;
⟨newNode.timeInsert:=getNextTimeStamp();⟩
if IS_MARKED(newNode.value) then
      newNode:=HelpDelete(newNode,0);
RELEASE_NODE(newNode);
return true;
```

Figure 7. Insert

```
// Local variables
prev,last,node1,node2 : pointer to Node

function DeleteMin():pointer to Node
D1   ⟨TraverseTimeStamps();⟩
D2   ⟨time:=getNextTimeStamp();⟩
D3   prev:=COPY_NODE(head);
D4   while true do
D5      node1:=ReadNext(&prev,0);
D6      if node1=tail then
D7         RELEASE_NODE(prev);
D8         RELEASE_NODE(node1);
D9         return NULL;
     retry:
D10     value:=node1.value;
D11     if not IS_MARKED(value) ⟨and
          compareTimeStamp(time,node1.timeInsert)>0⟩ then
D12        if CAS(&node1.value,value,
            GET_MARKED(value)) then
D13           node1.prev:=prev;
D14           break;
D15        else goto retry;
D16     else if IS_MARKED(value) then
D17        node1:=HelpDelete(node1,0);
D18     RELEASE_NODE(prev);
D19     prev:=node1;
D20  for i:=0 to node1.level-1 do
D21     repeat
D22        node2:=node1.next[i];
D23     until IS_MARKED(node2) or CAS(
         &node1.next[i],node2,GET_MARKED(node2));
D24  prev:=COPY_NODE(head);
D25  for i:=node1.level-1 to 0 step -1 do
D26     while true do
D27        if node1.next[i]=1 then break;
D28        last:=ScanKey(&prev,i,node1.key);
D29        RELEASE_NODE(last);
D30        if last$\neq$node1 or node1.next[i]=1 then break;
D31        if CAS(&prev.next[i],node1,
            GET_UNMARKED(node1.next[i])) then
D32           node1.next[i]:=1;
D33           break;
D34        if node1.next[i]=1 then break;
D35        Back-Off
D36  RELEASE_NODE(prev);
D37  RELEASE_NODE(node1);
D38  RELEASE_NODE(node1); /* Delete the node */
D39  return value;
```

Figure 8. DeleteMin

```
// Local variables
prev,last,node2 : pointer to Node

function HelpDelete(node:pointer to Node,
 level:integer):pointer to Node
H1   for i:=level to node.level-1 do
H2      repeat
H3         node2:=node.next[i];
H4      until IS_MARKED(node2) or CAS(
         &node.next[i],node2,GET_MARKED(node2));
H5   prev:=node.prev;
H6   if not prev or level $\geq$ prev.validLevel then
H7      prev:=COPY_NODE(head);
H8      for i:=maxLevel-1 to level step -1 do
H9         node2:=ScanKey(&prev,i,node.key);
H10        RELEASE_NODE(node2);
H11  else COPY_NODE(prev);
H12  while true do
H13     if node.next[level]=1 then break;
H14     last:=ScanKey(&prev,level,node.key);
H15     RELEASE_NODE(last);
H16     if last$\neq$node or node.next[level]=1 then break;
H17     if CAS(&prev.next[level],node,
         GET_UNMARKED(node.next[level])) then
H18        node.next[level]:=1;
H19        break;
H20     if node.next[level]=1 then break;
H21     Back-Off
H22  RELEASE_NODE(node);
H23  return prev;
```

Figure 9. HelpDelete

ure 7, starts in lines I5-I11 with a search phase to find the node after which the new node ($newNode$) should be inserted. This search phase starts from the head node at the highest level and traverses down to the lowest level until the correct node is found ($node1$). When going down one level, the last node traversed on that level is remembered ($savedNodes$) for later use (this is where we should insert the new node at that level). Now it is possible that there already exists a node with the same priority as of the new node, this is checked in lines I12-I24, the value of the old node ($node2$) is changed atomically with a CAS. Otherwise, in lines I25-I42 it starts trying to insert the new node starting with the lowest level and increasing up to the level of the new node. The next pointers of the nodes (to become previous) are changed atomically with a CAS. After the new node has been inserted at the lowest level, it is possible that it is deleted by a concurrent $DeleteMin$ operation before it has been inserted at all levels, and this is checked in lines I38 and I44.

The $DeleteMin$ operation, see Figure 8, starts from the head node and finds the first node ($node1$) in the list that does not have its deletion mark on the value set, see lines D3-D12. It tries to set this deletion mark in line D12 using the CAS primitive, and if it succeeds it also writes a valid pointer to the prev field of the node. This prev field is necessary in order to increase the performance of concurrent $HelpDelete$ operations, these operations otherwise would have to search for the previous node in order to complete the deletion. The next step is to mark the deletion bits of the next pointers in the node, starting with the lowest level and going upwards, using the CAS primitive in each step, see lines D20-D23. Afterwards in lines D24-D35 it starts the actual deletion by changing the next pointers of the previous node ($prev$), starting at the highest level and continuing downwards. The reason for doing the deletion in decreasing order of levels, is that concurrent search operations also start at the highest level and proceed downwards, in this way the concurrent search operations will sooner avoid traversing this node. The procedure performed by the $DeleteMin$ operation in order to change each next pointer of the previous node, is to first search for the previous node and then perform the CAS primitive until it succeeds.

The algorithm has been designed for pre-emptive as well as fully concurrent systems. In order to achieve the lock-free property (that at least one thread is doing progress) on pre-emptive systems, whenever a search operation finds a node that is about to be deleted, it calls the $HelpDelete$ operation and then proceeds searching from the previous node of the deleted. The $HelpDelete$ operation, see Figure 9, tries to fulfill the deletion on the current level and returns when it is completed. It starts in lines H1-H4 with setting the deletion mark on all next pointers in case they have not been set. In lines H5-H6 it checks if the node given in the prev field is valid for deletion on the current level, otherwise it searches for the correct node ($prev$) in lines H7-H10. The actual deletion of this node on the current level takes place in lines H12-H21. This operation might execute concurrently with the corresponding $DeleteMin$ operation, and therefore both operations synchronize with each other in lines D27, D30, D32, D34, H13, H16, H18 and H20 in order to avoid executing sub-operations that have already been performed.

In fully concurrent systems though, the helping strategy can downgrade the performance significantly. Therefore the algorithm, after a number of consecutive failed attempts to help concurrent $DeleteMin$ operations that hinders the progress of the current operation, puts the operation into back-off mode. When in back-off mode, the thread does nothing for a while, and in this way avoids disturbing the concurrent operations that might otherwise progress slower. The duration of the back-off is proportional to the number of threads, and for each consecutive entering of back-off mode during one operation invocation, the duration is increased exponentially.

4 Correctness

In this section we present the proof of our algorithm. We first prove that our algorithm is a linearizable one [5] and then we prove that it is lock-free. A set of definitions that will help us to structure and shorten the proof is first explained in this section. We start by defining the sequential semantics of our operations and then introduce two definitions concerning concurrency aspects in general.

Definition 1 *We denote with L_t the abstract internal state of a priority queue at the time t. L_t is viewed as a set of pairs $\langle p, v\rangle$ consisting of a unique priority p and a corresponding value v. The operations that can be performed on the priority queue are $Insert$ (I) and $DeleteMin$ (DM). The time t_1 is defined as the time just before the atomic execution of the operation that we are looking at, and the time t_2 is defined as the time just after the atomic execution of the same operation. The return value of $true_2$ is returned by an $Insert$ operation that has succeeded to update an existing node, the return value of $true$ is returned by an $Insert$ operation that succeeds to insert a new node. In the following expressions that defines the sequential semantics of our operations, the syntax is $S_1 : O_1, S_2$, where S_1 is the conditional state before the operation O_1, and S_2 is the resulting state after performing the corresponding operation:*

$$\langle p_1, \_\rangle \notin L_{t_1} : \mathbf{I_1}(\langle \mathbf{p_1}, \mathbf{v_1}\rangle) = \mathbf{true}, \\ \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \cup \{\langle \mathbf{p_1}, \mathbf{v_1}\rangle\} \quad (1)$$

$$\langle p_1, v_{1_1}\rangle \in L_{t_1} : \mathbf{I_1}(\langle \mathbf{p_1}, \mathbf{v_{1_2}}\rangle) = \mathbf{true_2}, \\ \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \setminus \{\langle \mathbf{p_1}, \mathbf{v_{1_1}}\rangle\} \cup \{\langle \mathbf{p_1}, \mathbf{v_{1_2}}\rangle\} \quad (2)$$

$$\langle p_1, v_1\rangle = \{\langle \min p, v\rangle | \langle p, v\rangle \in L_{t_1}\} \\ : \mathbf{DM_1}() = \langle \mathbf{p_1}, \mathbf{v_1}\rangle, \ \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \setminus \{\langle \mathbf{p_1}, \mathbf{v_1}\rangle\} \quad (3)$$

$$L_{t_1} = \emptyset : \mathbf{DM_1}() = \bot \quad (4)$$

Definition 2 *In a global time model each concurrent operation Op "occupies" a time interval $[b_{Op}, f_{Op}]$ on the linear time axis $(b_{Op} < f_{Op})$. The precedence relation (denoted by '$\rightarrow$') is a relation that relates operations of a possible execution, $Op_1 \rightarrow Op_2$ means that Op_1 ends before Op_2 starts. The precedence relation is a strict partial order. Operations incomparable under $\rightarrow$ are called* overlapping. *The overlapping relation is denoted by $\|$ and is commutative, i.e. $Op_1 \parallel Op_2$ and $Op_2 \parallel Op_1$. The precedence relation is extended to relate sub-operations of operations. Consequently, if $Op_1 \rightarrow Op_2$, then for any sub-operations op_1 and op_2 of Op_1 and Op_2, respectively, it holds that $op_1 \rightarrow op_2$. We also define the direct precedence relation $\rightarrow_d$, such that if $Op_1 \rightarrow_d Op_2$, then $Op_1 \rightarrow Op_2$ and moreover there exists no operation Op_3 such that $Op_1 \rightarrow Op_3 \rightarrow Op_2$.*

Definition 3 *In order for an implementation of a shared concurrent data object to be linearizable [5], for every concurrent execution there should exist an equal (in the sense of the effect) and valid (i.e. it should respect the semantics of the shared data object) sequential execution that respects the partial order of the operations in the concurrent execution.*

Next we are going to study the possible concurrent executions of our implementation. First we need to define the interpretation of the abstract internal state of our implementation.

Definition 4 *The pair $\langle p, v\rangle$ is present ($\langle p, v\rangle \in L$) in the abstract internal state L of our implementation, when there is a next pointer from a present node on the lowest level of the Skiplist that points to a node that contains the pair $\langle p, v\rangle$, and this node is not marked as deleted with the mark on the value.*

Lemma 1 *The definition of the abstract internal state for our implementation is consistent with all concurrent operations examining the state of the priority queue.*

Proof: As the next and value pointers are changed using the CAS operation, we are sure that all threads see the same state of the Skiplist, and therefore all changes of the abstract internal state seems to be atomic. □

Definition 5 *The decision point of an operation is defined as the atomic statement where the result of the operation is finitely decided, i.e. independent of the result of any sub-operations proceeding the decision point, the operation will have the same result. We define the state-read point of an operation to be the atomic statement where a sub-state of the priority queue is read, and this sub-state is the state on which the decision point depends. We also define the state-change point as the atomic statement where the operation changes the abstract internal state of the priority queue after it has passed the corresponding decision point.*

We will now show that all of these points conform to the very same statement, i.e. the linearizability point.

Lemma 2 *An $Insert$ operation which succeeds ($I(\langle p, v\rangle) = true$), takes effect atomically at one statement.*

Proof: The decision point for an $Insert$ operation which succeeds ($I(\langle p, v\rangle) = true$), is when the CAS sub-operation in line I27 (see Figure 7) succeeds, all following CAS sub-operations will eventually succeed, and the $Insert$ operation will finally return $true$. The state of the list (L_{t_1}) directly before the passing of the decision point must have been $\langle p, \_\rangle \notin L_{t_1}$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle p, v\rangle \in L_{t_2}$. □

Lemma 3 *An $Insert$ operation which updates ($I(\langle p, v\rangle) = true_2$), takes effect atomically at one statement.*

Proof: The decision point for an $Insert$ operation which updates ($I(\langle p, v\rangle) = true_2$), is when the CAS will succeed in line I14. The state of the list (L_{t_1}) directly before passing the decision point must have been $\langle p, \_\rangle \in L_{t_1}$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle p, v\rangle \in L_{t_3}$. □

Lemma 4 *A $DeleteMin$ operation which succeeds ($D() = \langle p, v\rangle$), takes effect atomically at one statement.*

Proof: The decision point for an $DeleteMin$ operation which succeeds ($D() = \langle p, v\rangle$) is when the CAS sub-operation in line D12 (see Figure 8) succeeds. The state of the list (L_t) directly before passing of the decision point must have been $\langle p, v\rangle \in L_t$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle p, \_\rangle \notin L_t$. □

Lemma 5 *A $DeleteMin$ operations which fails ($D() = \bot$), takes effect atomically at one statement.*

Proof: The decision point and also the state-read point for an $DeleteMin$ operations which fails ($D() = \bot$), is when the hidden read sub-operation of the $ReadNext$ sub-operation in line D5 successfully reads the next pointer on lowest level that equals the tail node. The state of the list (L_t) directly before the passing of the state-read point must have been $L_t = \emptyset$. □

Definition 6 *We define the relation $\Rightarrow$ as the total order and the relation $\Rightarrow_d$ as the direct total order between all operations in the concurrent execution. In the following formulas, $E_1 \Longrightarrow E_2$ means that if E_1 holds then E_2 holds as well, and $\oplus$ stands for exclusive or (i.e. $a \oplus b$ means $(a \vee b) \wedge \neg(a \wedge b)$):*

$$\mathbf{Op_1} \rightarrow_{\mathbf{d}} \mathbf{Op_2}, \nexists \mathbf{Op_3}.\mathbf{Op_1} \Rightarrow_{\mathbf{d}} \mathbf{Op_3}, \\ \nexists \mathbf{Op_4}.\mathbf{Op_4} \Rightarrow_{\mathbf{d}} \mathbf{Op_2} \Longrightarrow \mathbf{Op_1} \Rightarrow_{\mathbf{d}} \mathbf{Op_2} \tag{5}$$

$$\mathbf{Op_1} \parallel \mathbf{Op_2} \Longrightarrow \mathbf{Op_1} \Rightarrow_{\mathbf{d}} \mathbf{Op_2} \oplus \mathbf{Op_2} \Rightarrow_{\mathbf{d}} \mathbf{Op_1} \tag{6}$$

$$\mathbf{Op_1} \Rightarrow_{\mathbf{d}} \mathbf{Op_2} \Longrightarrow \mathbf{Op_1} \Rightarrow \mathbf{Op_2} \tag{7}$$

$$\mathbf{Op_1} \Rightarrow \mathbf{Op_2}, \mathbf{Op_2} \Rightarrow \mathbf{Op_3} \Longrightarrow \mathbf{Op_1} \Rightarrow \mathbf{Op_3} \tag{8}$$

Lemma 6 *The operations that are directly totally ordered using formula 5, form an equivalent valid sequential execution.*

Proof: If the operations are assigned their direct total order ($Op_1 \Rightarrow_d Op_2$) by formula 5 then also the decision, state-read and the state-change points of Op_1 is executed before the respective points of Op_2. In this case the operations semantics behave the same as in the sequential case, and therefore all possible executions will then be equivalent to one of the possible sequential executions. □

Lemma 7 *The operations that are directly totally ordered using formula 6 can be ordered unique and consistent, and form an equivalent valid sequential execution.*

Proof: Assume we order the overlapping operations according to their decision points. As the state before as well as after the decision points is identical to the corresponding state defined in the semantics of the respective sequential operations in formulas 1 to 4, we can view the operations as occurring at the decision point. As the decision points consist of atomic operations and are therefore ordered in time, no decision point can occur at the very same time as any other decision point, therefore giving a unique and consistent ordering of the overlapping operations. □

Lemma 8 *With respect to the retries caused by synchronization, one operation will always do progress regardless of the actions by the other concurrent operations.*

Proof: We now examine the possible execution paths of our implementation. There are several potentially unbounded loops that can delay the termination of the operations. We call these loops retry-loops. If we omit the conditions that are because of the operations semantics (i.e. searching for the correct position etc.), the retry-loops take place when sub-operations detect that a shared variable has changed value. This is detected either by a subsequent read sub-operation or a failed CAS. These shared variables are only changed concurrently by other CAS sub-operations. According to the definition of CAS, for any number of concurrent CAS sub-operations, exactly one will succeed. This means that for any subsequent retry, there must be one CAS that succeeded. As this succeeding CAS will cause its retry loop to exit, and our implementation does not contain any cyclic dependencies between retry-loops that exit with CAS, this means that the corresponding $Insert$ or $DeleteMin$ operation will progress. Consequently, independent of any number of concurrent operations, one operation will always progress. □

Theorem 1 *The algorithm implements a lock-free and linearizable priority queue.*

Proof: Following from Lemmas 6 and 7 and using the direct total order we can create an identical (with the same semantics) sequential execution that preserves the partial order of the operations in a concurrent execution. Following from Definition 3, the implementation is therefore linearizable. As the semantics of the operations are basically the same as in the Skiplist [11], we could use the corresponding proof of termination. This together with Lemma 8 and that the state is only changed at one atomic statement (Lemmas 1,2,3,4,5), gives that our implementation is lock-free. □

5 Experiments

In our experiments each concurrent thread performs 10000 sequential operations, whereof the first 100 or 1000 operations are $Insert$ operations, and the remaining operations are randomly chosen with a distribution of 50% $Insert$ operations versus 50% $DeleteMin$ operations. The key values of the inserted nodes are randomly chosen between 0 and $1000000 * n$, where n is the number of threads. Each experiment is repeated 50 times, and an average execution time for each experiment is estimated. Exactly the same sequential operations are performed for all different implementations compared. Besides our implementation, we also performed the same experiment with two lock-based implementations. These are; 1) the implementation using multiple locks and Skiplists by Lotan *et al.* [9] which is the most recently claimed to be one of the most efficient concurrent priority queues existing, and 2) the heap-based implementation using multiple locks by Hunt *et al.* [7]. All lock-based implementations are based on simple spin-locks using the TAS atomic primitive. A clean-cache operation is performed just before each sub-experiment. All implementations are written in C and compiled with the highest optimization level, except from the atomic primitives, which are written in assembler.

The experiments were performed using different number of threads, varying from 1 to 30. To get a highly pre-emptive environment, we performed our experiments on a Compaq dual-processor Pentium II 450 MHz PC running Linux. A set of experiments was also performed on a Sun Solaris system with 4 processors. In order to evaluate our algorithm with full concurrency we also used a SGI Origin 2000 195 MHz system running Irix with 64 processors. The results from these experiments are shown in Figure 10 together with a close-up view of the Sun experiment. The average execution time is drawn as a function of the number of threads.

From the results we can conclude that all of the implementations scale similarly with respect to the average size of the queue. The implementation by Lotan and Shavit [9] scales linearly with respect to increasing number of threads when having full concurrency, although when exposed to pre-emption its performance decreases very rapidly; already with 4 threads the performance decreased with over 20

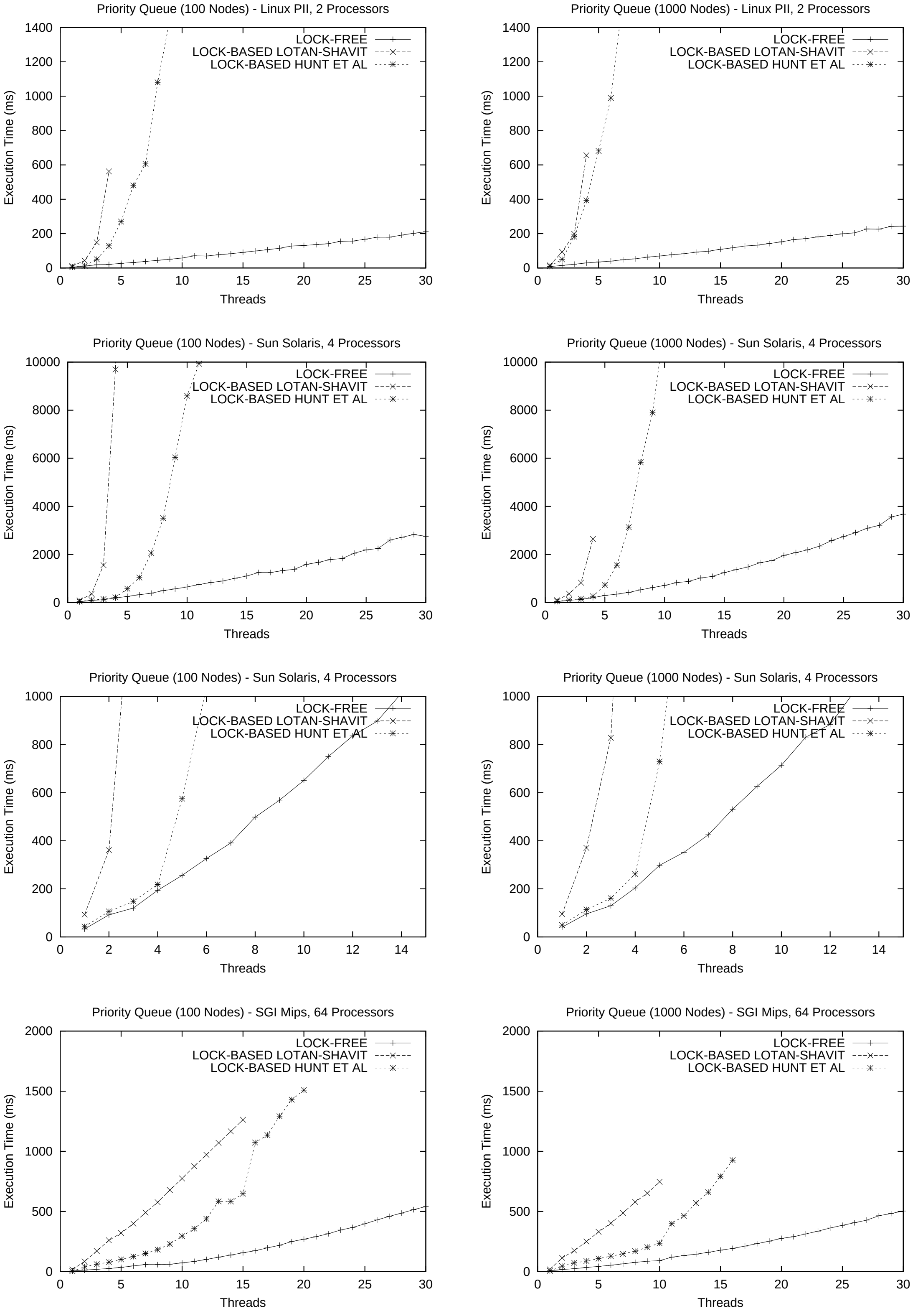

Figure 10. Experiment with priority queues and high contention, initialized with 100 respective 1000 nodes

times. We must point out here that the implementation by Lotan and Shavit is designed for a large (i.e. 256) number of processors. The implementation by Hunt *et al.* [7] shows better but similar behavior. Because of this behavior we decided to run the experiments for these two implementations only up to a certain number of threads to avoid getting timeouts. Our lock-free implementation scales best compared to all other involved implementations, having best performance already with 3 threads, independently if the system is fully concurrent or involves pre-emptions.

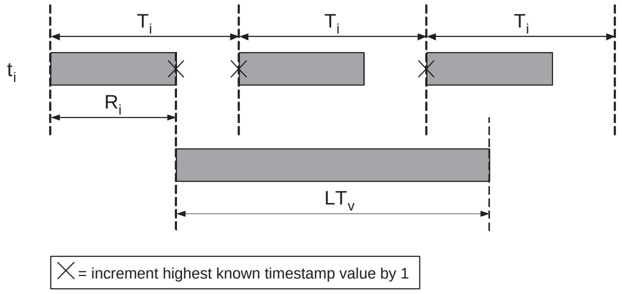

Figure 11. Maximum timestamp increasement estimation - worst case scenario

6 Extended Algorithm

When we have concurrent $Insert$ and $DeleteMin$ operations we might want to have certain real-time properties of the semantics of the $DeleteMin$ operation, as expressed in [9]. The $DeleteMin$ operation should only return items that have been inserted by an $Insert$ operation that finished before the $DeleteMin$ operation started. To ensure this we are adding timestamps to each node. When the node is fully inserted its timestamp is set to the current time. Whenever the $DeleteMin$ operation is invoked it first checks the current time, and then discards all nodes that have a timestamp that is after this time. In the code of the implementation (see Figures 6,7,8 and 9), the additional statements that involve timestamps are marked within the "$\langle$" and "$\rangle$" symbols. The function $getNextTimeStamp$, see Figure 14, creates a new timestamp. The function $compareTimeStamp$, see Figure 14, compares if the first timestamp is less, equal or higher than the second one and returns the values -1,0 or 1, respectively.

As we are only using the timestamps for relative comparisons, we do not need real absolute time, only that the timestamps are monotonically increasing. Therefore we can implement the time functionality with a shared counter, the synchronization of the counter is handled using CAS. However, the shared counter usually has a limited size (i.e. 32 bits) and will eventually overflow. Therefore the values of the timestamps have to be recycled. We will do this by exploiting information that are available in real-time systems, with a similar approach as in [15].

We assume that we have n periodic tasks in the system, indexed $\tau_1 ... \tau_n$. For each task τ_i we will use the standard notations T_i, C_i, R_i and D_i to denote the period (i.e. $\min$ period for sporadic tasks), worst case execution time, worst case response time and deadline, respectively. The deadline of a task is less or equal to its period.

For a system to be safe, no task should miss its deadlines, i.e. $\forall i \mid R_i \leq D_i$.

For a system scheduled with fixed priority, the response time for a task in the initial system can be calculated using the standard response time analysis techniques [1]. If we with B_i denote the blocking time (the time the task can be delayed by lower priority tasks) and with $hp(i)$ denote the set of tasks with higher priority than task τ_i, the response time R_i for task τ_i can be formulated as:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad (9)$$

The summand in the above formula gives the time that task τ_i may be delayed by higher priority tasks. For systems scheduled with dynamic priorities, there are other ways to calculate the response times [1].

Now we examine some properties of the timestamps that can exist in the system. Assume that all tasks call either the $Insert$ or $DeleteMin$ operation only once per iteration. As each call to $getNextTimeStamp$ will introduce a new timestamp in the system, we can assume that every task invocation will introduce one new timestamp. This new timestamp has a value that is the previously highest known value plus one. We assume that the tasks always execute within their response times R with arbitrary many interruptions, and that the execution time C is comparably small. This means that the increment of highest timestamp respective the write to a node with the current timestamp can occur anytime within the interval for the response time. The maximum time for an $Insert$ operation to finish is the same as the response time R_i for its task τ_i. The minimum time between two index increments is when the first increment is executed at the end of the first interval and the next increment is executed at the very beginning of the second interval, i.e. $T_i - R_i$. The minimum time between the subsequent increments will then be the period T_i. If we denote with LT_v the maximum life-time that the timestamp with value v exists in the system, the worst case scenario in respect of growth of timestamps is shown in Figure 11.

The formula for estimating the maximum difference in value between two existing timestamps in any execution becomes as follows:

$$MaxTag = \sum_{i=0}^{n} \left(\left\lceil \frac{\max_{v\in\{0..\infty\}} LT_v}{T_i} \right\rceil + 1 \right) \qquad (10)$$

Now we have to bound the value of $\max_{v\in\{0..\infty\}} LT_v$. When comparing timestamps, the absolute value of these are not important, only the relative values. Our method is that we continuously traverse the nodes and replace outdated timestamps with a newer timestamp that has the same comparison result. We traverse and check the nodes at the rate of one step to the right for every invocation of an $Insert$ or $DeleteMin$ operation. With outdated timestamps we define timestamps that are older (i.e. lower) than any timestamp value that is in use by any running $DeleteMin$ operation. We denote with $AncientVal$ the maximum difference that we allow between the highest known timestamp value and the timestamp value of a node, before we call this timestamp outdated.

$$AncientVal = \sum_{i=0}^{n} \left\lceil \frac{\max_j R_j}{T_i} \right\rceil \qquad (11)$$

If we denote with t_{ancient} the maximum time it takes for a timestamp value to be outdated counted from its first occurrence in the system, we get the following relation:

$$AncientVal = \sum_{i=0}^{n} \left\lfloor \frac{t_{\text{ancient}}}{T_i} \right\rfloor > \sum_{i=0}^{n} \left(\frac{t_{\text{ancient}}}{T_i} \right) - n \qquad (12)$$

$$t_{\text{ancient}} < \frac{AncientVal + n}{\sum_{i=0}^{n} \frac{1}{T_i}} \qquad (13)$$

Now we denote with t_{traverse} the maximum time it takes to traverse through the whole list from one position and getting back, assuming the list has the maximum size N.

$$N = \sum_{i=0}^{n} \left\lfloor \frac{t_{\text{traverse}}}{T_i} \right\rfloor > \sum_{i=0}^{n} \left(\frac{t_{\text{traverse}}}{T_i} \right) - n \qquad (14)$$

$$t_{\text{traverse}} < \frac{N + n}{\sum_{i=0}^{n} \frac{1}{T_i}} \qquad (15)$$

The worst-case scenario is that directly after the timestamp of one node gets traversed, it gets outdated. Therefore we get:

$$\max_{v\in\{0..\infty\}} LT_v = t_{\text{ancient}} + t_{\text{traverse}} \qquad (16)$$

Putting all together we get:

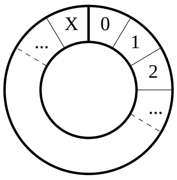

Figure 12. Timestamp value recycling

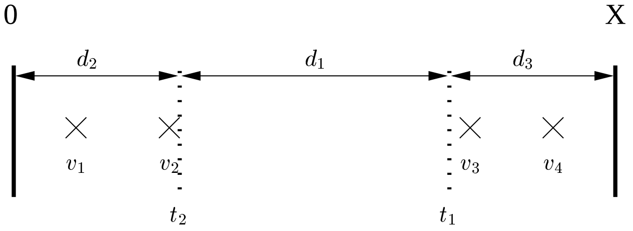

Figure 13. Deciding the relative order between reused timestamps

$$MaxTag < \sum_{i=0}^{n} \left(\left\lceil \frac{N + 2n + \sum_{k=0}^{n} \left\lceil \frac{\max_j R_j}{T_k} \right\rceil}{T_i \sum_{l=0}^{n} \frac{1}{T_l}} \right\rceil + 1 \right) \qquad (17)$$

The above equation gives us a bound on the length of the "window" of active timestamps for any task in any possible execution. In the unbounded construction the tasks, by producing larger timestamps every time they slide this window on the $[0, \ldots, \infty]$ axis, always to the right. The approach now is instead of sliding this window on the set $[0, \ldots, \infty]$ from left to right, to cyclically slide it on a $[0, \ldots, X]$ set of consecutive natural numbers, see figure 12. Now at the same time we have to give a way to the tasks to identify the order of the different timestamps because the order of the physical numbers is not enough since we are re-using timestamps. The idea is to use the bound that we have calculated for the span of different active timestamps. Let us then take a task that has observed v_i as the lowest timestamp at some invocation τ. When this task runs again as τ', it can conclude that the active timestamps are going to be between v_i and $(v_i + MaxTag) \mod X$. On the other hand we should make sure that in this interval $[v_i, \ldots, (v_i + MaxTag) \mod X]$ there are no old timestamps. By looking closer to equation 10 we can conclude that all the other tasks have written values to their registers with timestamps that are at most $MaxTag$ less than v_i at the time that τ wrote the value v_i. Consequently if we use an interval that has double the

```
// Global variables
timeCurrent: integer
checked: pointer to Node
// Local variables
time,newtime,safeTime: integer
current,node,next: pointer to Node

function compareTimeStamp(time1:integer,
 time2:integer):integer
if time1=time2 then return 0;
if time2=MAX_TIME then return -1;
if time1$>$time2 and (time1-time2)$\leq$MAX_TAG or
 time1$<$time2 and (time1-time2+MAX_TIME)
 $\leq$MAX_TAG then return 1;
else return -1;

function getNextTimeStamp():integer
repeat
    time:=timeCurrent;
    if (time+1)$\neq$MAX_TIME then newtime:=time+1;
    else newtime:=0;
until CAS(&timeCurrent,time,newtime);
return newtime;

procedure TraverseTimeStamps()
safeTime:=timeCurrent;
if safeTime$\geq$ANCIENT_VAL then
    safeTime:=safeTime-ANCIENT_VAL;
else safeTime:=safeTime+MAX_TIME-ANCIENT_VAL;
while true do
    node:=READ_NODE(checked);
    current:=node;
    next:=ReadNext(&node,0);
    RELEASE_NODE(node);
    if compareTimeStamp(safeTime,next.timeInsert)$>$0 then
        next.timeInsert:=safeTime;
    if CAS(&checked,current,next) then
        RELEASE_NODE(current);
        break;
    RELEASE_NODE(next);
```

Figure 14. Creation, comparison, traversing and updating of bounded timestamps.

size of $MaxTag$, τ' can conclude that old timestamps are all on the interval $[(v_i - MaxTag) \mod X, \ldots, v_i]$.

Therefore we can use a timestamp field with double the size of the maximum possible value of the timestamp.

$$TagFieldSize = MaxTag * 2$$
$$TagFieldBits = \lceil \log_2 TagFieldSize \rceil$$

In this way τ' will be able to identify that v_1, v_2, v_3, v_4 (see figure 13) are all new values if $d_2 + d_3 < MaxTag$ and can also conclude that:

$$v_3 < v_4 < v_1 < v_2$$

The mechanism that will generate new timestamps in a cyclical order and also compare timestamps is presented in Figure 14 together with the code for traversing the nodes. Note that the extra properties of the priority queue that are achieved by using timestamps are not complete with respect to the $Insert$ operations that finishes with an update. These update operations will behave the same as for the standard version of the implementation.

Besides from real-time systems, the presented technique can also be useful in non real-time systems as well. For example, consider a system of $n = 10$ threads, where the minimum time between two invocations would be $T = 10$ ns, and the maximum response time $R = 1000000000$ ns (i.e. after 1 s we would expect the thread to have crashed). Assuming a maximum size of the list $N = 10000$, we will have a maximum timestamp difference $MaxTag <$ 1000010030, thus needing 31 bits. Given that most systems have 32-bit integers and that many modern systems handle 64 bits as well, it implies that this technique is practical for also non real-time systems.

7 Conclusions

We have presented a lock-free algorithmic implementation of a concurrent priority queue. The implementation is based on the sequential Skiplist data structure and builds on top of it to support concurrency and lock-freedom in an efficient and practical way. Compared to the previous attempts to use Skiplists for building concurrent priority queues our algorithm is lock-free and avoids the performance penalties that come with the use of locks. Compared to the previous lock-free/wait-free concurrent priority queue algorithms, our algorithm inherits and carefully retains the basic design characteristic that makes Skiplists practical: simplicity. Previous lock-free/wait-free algorithms did not perform well because of their complexity, furthermore they were often based on atomic primitives that are not available in today's systems.

We compared our algorithm with some of the most efficient implementations of priority queues known. Experiments show that our implementation scales well, and with 3 threads or more our implementation outperforms the corresponding lock-based implementations, for all cases on both fully concurrent systems as well as with pre-emption.

We believe that our implementation is of highly practical interest for multi-threaded applications. We are currently incorporating it into the NOBLE [16] library.

References

[1] N.C. AUDSLEY, A. BURNS, R.I. DAVIS, K.W. TINDELL, A.J. WELLINGS. Fixed Priority Pre-emptive Scheduling: An Historical Perspective. *Real-Time Systems*, Vol. 8, Num. 2/3, pp. 129-154, 1995.

[2] G. BARNES. Wait-Free Algorithms for Heaps. *Technical Report*, Computer Science and Engineering, University of Washington, Feb. 1992.

[3] M. GRAMMATIKAKIS, S. LIESCHE. Priority queues and sorting for parallel simulation. *IEEE Transactions on Software Engineering*, SE-26 (5), pp. 401-422, 2000.

[4] T. L. HARRIS. A Pragmatic Implementation of Non-Blocking Linked Lists. *Proceedings of the 15th International Symposium of Distributed Computing*, Oct. 2001.

[5] M. HERLIHY, J. WING. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[6] M. HERLIHY. Wait-Free Synchronization. *ACM TOPLAS*, Vol. 11, No. 1, pp. 124-149, Jan. 1991.

[7] G. HUNT, M. MICHAEL, S. PARTHASARATHY, M. SCOTT. An Efficient Algorithm for Concurrent Priority Queue Heaps. *Information Processing Letters*, 60(3), pp. 151-157, Nov. 1996.

[8] A. ISRAELI, L. RAPPAPORT. Efficient wait-free implementation of a concurrent priority queue. *7th Intl Workshop on Distributed Algorithms '93*, Lecture Notes in Computer Science 725, Springer Verlag, pp 1-17, Sept. 1993.

[9] I. LOTAN, N. SHAVIT. Skiplist-Based Concurrent Priority Queues. *International Parallel and Distributed Processing Symposium*, 2000.

[10] M. MICHAEL, M. SCOTT. Correction of a Memory Management Method for Lock-Free Data Structures. *Computer Science Dept., University of Rochester*, 1995.

[11] W. PUGH. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, vol.33, no.6, pp. 668-76, June 1990.

[12] R. RAJKUMAR. Real-Time Synchronization Protocols for Shared Memory Multiprocessors. *10th International Conference on Distributed Computing Systems*, pp. 116-123, 1990.

[13] L. SHA AND R. RAJKUMAR, J. LEHOCZKY. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers* Vol. 39, 9, pp. 1175-1185, Sep. 1990.

[14] A. SILBERSCHATZ, P. GALVIN. Operating System Concepts. *Addison Wesley*, 1994.

[15] H. SUNDELL, P. TSIGAS. Space Efficient Wait-Free Buffer Sharing in Multiprocessor Real-Time Systems Based on Timing Information. *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applicatons (RTCSA 2000)*, pp. 433-440, IEEE press, 2000.

[16] H. SUNDELL, P. TSIGAS. NOBLE: A Non-Blocking Inter-Process Communication Library. *Proceedings of the 6th Workshop on Languages, Compilers and Runtime Systems for Scalable Computers (LCR'02)*, Lecture Notes in Computer Science, Springer Verlag, 2002.

[17] P. TSIGAS, Y. ZHANG. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. *Proceedings of the international conference on Measurement and modeling of computer systems (SIGMETRICS 2001)*, pp. 320-321, ACM Press, 2001.

[18] P. TSIGAS, Y. ZHANG. Integrating Non-blocking Synchronisation in Parallel Applications: Performance Advantages and Methodologies. *Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP '02)*, ACM Press, 2002.

[19] J. D. VALOIS. Lock-Free Data Structures. *PhD. Thesis, Rensselaer Polytechnic Institute, Troy, New York*, 1995.