Scalable and Lock-Free Concurrent Dictionaries

Håkan Sundell
Computing Science
Chalmers University of Technology
Göteborg, Sweden
phs@cs.chalmers.se

Philippas Tsigas
Computing Science
Chalmers University of Technology
Göteborg, Sweden
tsigas@cs.chalmers.se

Categories and Subject Descriptors

E.1 [**Data Structures**]: Distributed Data Structures

General Terms

Algorithms, Performance, Reliability, Experimentation

Keywords

Concurrent, non-blocking, dictionary, shared memory

ABSTRACT

We present an efficient and practical lock-free implementation of a concurrent dictionary that is suitable for both fully concurrent (large multi-processor) systems as well as pre-emptive (multi-process) systems. Many algorithms for concurrent dictionaries 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. Our algorithm is based on the randomized sequential list structure called Skiplist, and implements the full set of operations on a dictionary that is suitable for practical settings. In our performance evaluation we compare our algorithm with the most efficient non-blocking implementation of dictionaries known. The experimental results clearly show that our algorithm outperforms the other lock-free algorithm for dictionaries with realistic sizes, both on fully concurrent as well as pre-emptive systems.

1. INTRODUCTION

Dictionaries (also called sets) are fundamental data structures. From the operating system level to the user application level, they are frequently used as basic components. Consequently, the design of efficient implementations of dictionaries is a research area that has been extensively researched. A dictionary supports five operations, the *Insert*, the *FindKey*, the *DeleteKey*, the *FindValue* and the *DeleteValue* operation. The abstract definition of a dictionary is a set of key-value pairs, where the key is an unique integer associated with a value. The *Insert* operation inserts a new key-value pair into the dictionary and the *FindKey*/*DeleteKey* operation finds/removes and returns the value of the key-value pair with the specified key that was in the dictionary. The *FindValue*/*DeleteValue* operation finds / removes and returns the key of the key-value pair with the specified value that was in the dictionary.

To ensure consistency of a shared data object in a concurrent environment, the most common method is mutual exclusion, i.e. some form of locking. Mutual exclusion degrades the system's overall performance [12] 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. Mutual exclusion can also cause deadlocks, priority inversion (which can be solved efficiently on uni-processors [11] with the cost of more difficult analysis, although not as efficient on multiprocessor systems [10]) and even starvation.

Researchers have addressed these problems by proposing 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 some operations could cause some specific other operations 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 [4] 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 its own steps, regardless of the actions of the concurrent operations. Non-blocking algorithms have been shown to be of big practical importance [16] to high-performance applications, and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [13].

There exist several algorithms and implementations of concurrent dictionaries. The majorities of the algorithms are lock-based, constructed with either a single lock on top of a sequential algorithm, or specially constructed algorithms using multiple locks, where each lock protects a part of the shared data structure. However, most lock-based algorithms [2] are based on the theoretical PRAM model which is shown to be unrealistic [1]. As the time complexity of the search operation of a dictionary is significant, most algorithms are based on tree or heap structures as well as tree-like structures as the Skiplist [9]. Previous non-blocking dictionaries are though based on arrays or ordered lists as done by Valois [17]. The path of using concurrent ordered lists for constructing non-blocking dictionaries has been improved by Harris [3], and lately [6] presented a significant improvement by using a new memory management method [7]. However, Valois [17] presented an incomplete idea of how to design a concurrent Skiplist.

One common problem with many algorithms for concurrent dictionaries is the lack of precise defined semantics of the operations. Previously known non-blocking dictionaries only implements a limited set of operations, disregarding the *FindValue* and *DeleteValue* 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 dictionary that is designed for efficient use in both pre-emptive as well as in fully concurrent environments. Inspired by the incomplete attempt by Valois [17], the algorithm is based on the randomized Skiplist [9] data structure. 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 we give a proof that our implementation is lock-free and linearizable.

Concurrent dictionaries are often used as building blocks for concurrent hash tables, where each branch (or bucket) of the hash table is represented by a dictionary. In an optimal setting, the average size of each branch is comparably low, i.e. less than 10 nodes, as in [6]. However, in practical settings the average size of each branch can vary significantly. For example, a hash table can be used to represent the words of a book, where each branch contains the words that begin with a certain letter. Therefore it is not unreasonable to expect dictionaries with sizes of several thousands nodes.

We have performed experiments that compare the performance of our algorithm with one of the most efficient implementations of non-blocking dictionaries known [6]. As the previous algorithm did not implement the full set of operations of our dictionary, we also performed experiments with the full set of operations, compared with a simple lock-based Skiplist implementation. Experiments were performed on two different platforms, consisting of a multiprocessor system using different operating systems and equipped with either 2 or 64 processors. Our results show that our algorithm outperforms the other lock-free implementation with realistic sizes and number of threads, in both highly pre-emptive as well as in fully concurrent environments.

The rest of the paper is organized as follows. In Section 2 we describe the type of 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 superior performance for our implementation is presented in Section 5. We conclude the paper with Section 6.

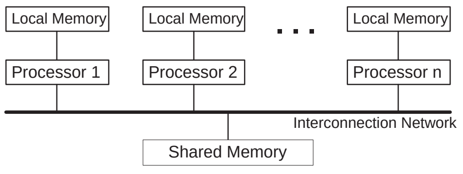

Figure 1: Shared Memory Multiprocessor System Structure

2. SYSTEM DESCRIPTION

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; some processors can have slower access than the others.

3. ALGORITHM

The algorithm is an extension and modification of the parallel Skiplist data structure presented in [14]. The sequential Skiplist data structure which was invented by Pugh [9], uses randomization and has a probabilistic time complexity of $O(\log N)$ 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. The maximum height (i.e. the maximum number of next pointers) of the data structure is $\log N$. 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 dictionary, every node contains a key and its corresponding value. The nodes are ordered in respect of key (which has to be unique for each node), the nodes with lowest keys are located first in the list. The fields of each node item are described in Figure 2 as it is used in this implementation. In all code figures in this section, arrays are indexed starting from 0.

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). See the full version of this paper [15] for a brief description of those primitives.

3.1 Memory Management

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

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

// Global variables
head,tail : pointer to Node
// Local variables (for all functions/procedures)
node1,node2,newNode,savedNodes[maxlevel+1] : pointer to Node
prev,last,stop : pointer to Node
key1,key2,step,jump,version,version2: integer

function CreateNode(level:integer, key:integer,
 value:pointer to word):pointer to Node
C1      node:=MALLOC_NODE();
C2      node.prev:=NULL;
C3      node.validLevel:=0;
C4      node.level:=level;
C5      node.key:=key;
C6      node.value:=value;
C7      return node;

procedure ReleaseReferences(node:pointer to Node)
R1      node.validLevel:=0;
R2      if node.prev then
R3          prev:=node.prev;
R4          node.prev:=NULL;
R5          RELEASE_NODE(prev);

function SearchLevel(last:pointer to pointer to Node,
 level:integer, key:integer): pointer to Node
S1      node1:=*last;
S2      stop:=NULL;
S3      while true do
S4          node2:=GET_UNMARKED(node1.next[level]);
S5          if node2=NULL then
S6              if node1=*last then
S7                  *last:=HelpDelete(*last,level);
S8              node1:=*last;
S9          else if node2.key≥key then
S10             COPY_NODE(node1);
S11             if (node1.validLevel¿level or node1=*last or node1=stop)
                        and node1.key¡key and node1.key≥(*last).key then
S12                 if node1.validLevel≤level then
S13                     RELEASE_NODE(node1);
S14                     node1:=COPY_NODE(*last);
S15                     node2:=ScanKey(&node1,level,key);
S16                     RELEASE_NODE(node2);
S17                 return node1;
S18             RELEASE_NODE(node1);
S19             stop:=node1;
S20             if IS_MARKED((*last).value) then
S21                 *last:=HelpDelete(*last,level);
S22             node1:=*last;
S23         else if node2.key≥(*last).key then
S24             node1:=node2;
S25         else
S26             if IS_MARKED((*last).value) then
S27                 *last:=HelpDelete(*last,level);
S28             node1:=*last;

function Insert(key:integer, value:pointer to word):boolean
I1      Choose level randomly according to the Skiplist distribution
I2      newNode:=CreateNode(level,key,value);
I3      COPY_NODE(newNode);
I4      savedNodes[maxLevel]:=head;
I5      for i:=maxLevel-1 to 0 step -1 do
I6          savedNodes[i]:=SearchLevel(&savedNodes[i+1],i,key);
I7          if maxLevel-1>i≥level-1 then
             RELEASE_NODE(savedNodes[i+1]);
I8      node1:=savedNodes[0];
I9      while true do
I10         node2:=ScanKey(&node1,0,key);
I11         value2:=node2.value;
I12         if not IS_MARKED(value2) and node2.key=key then
I13             if CAS(&node2.value,value2,value) then
I14                 RELEASE_NODE(node1);
I15                 RELEASE_NODE(node2);
I16                 for i:=1 to level-1 do
I17                     RELEASE_NODE(savedNodes[i]);
I18                 RELEASE_NODE(newNode);
I19                 RELEASE_NODE(newNode);
I20                 return $true_2$;
I21             else
I22                 RELEASE_NODE(node2);
I23                 continue;
I24         newNode.next[0]:=node2;
I25         RELEASE_NODE(node2);
I26         if CAS(&node1.next[0],node2,newNode) then
I27             RELEASE_NODE(node1);
I28             break;
I29         Back-Off
I30     newNode.version:=newNode.version+1;
I31     newNode.validLevel:=1;
I32     for i:=1 to level-1 do
I33         node1:=savedNodes[i];
I34         while true do
I35             node2:=ScanKey(&node1,i,key);
I36             newNode.next[i]:=node2;
I37             RELEASE_NODE(node2);
I38             if IS_MARKED(newNode.value) then
I39                 RELEASE_NODE(node1);
I40                 break;
I41             if CAS(&node1.next[i],node2,newNode) then
I42                 newNode.validLevel:=i+1;
I43                 RELEASE_NODE(node1);
I44                 break;
I45             Back-Off
I46     if IS_MARKED(newNode.value) then
         newNode:=HelpDelete(newNode,0);
I47     RELEASE_NODE(newNode);
I48     return true;

function FindKey(key: integer):pointer to word
F1      last:=COPY_NODE(head);
F2      for i:=maxLevel-1 to 0 step -1 do
F3          node1:=SearchLevel(&last,i,key);
F4          RELEASE_NODE(last);
F5          last:=node1;
F6      node2:=ScanKey(&last,0,key);
F7      RELEASE_NODE(last);
F8      value:=node2.value;
F9      if node2.key≠key or IS_MARKED(value) then
F10         RELEASE_NODE(node2);
F11         return NULL;
F12     RELEASE_NODE(node2);
F13     return value;

function DeleteKey(key: integer):pointer to word
        return Delete(key,false,NULL);
function Delete(key: integer, delval:boolean,
 value:pointer to word):pointer to word
D1      savedNodes[maxLevel]:=head;
D2      for i:=maxLevel-1 to 0 step -1 do
D3          savedNodes[i]:=SearchLevel(&savedNodes[i+1],i,key);
D4      node1:=ScanKey(&savedNodes[0],0,key);
D5      while true do
D6          if not delval then value:=node1.value;
D7          if node1.key=key and (not delval or node1.value=value)
             and not IS_MARKED(value) then
D8              if CAS(&node1.value,value,GET_MARKED(value)) then
D9                  node1.prev:=COPY_NODE(savedNodes[(node1.level-1)/2]);
D10                 break;
D11             else continue;
D12         RELEASE_NODE(node1);
D13         for i:=0 to maxLevel-1 do
D14             RELEASE_NODE(savedNodes[i]);
D15         return NULL;
D16     for i:=0 to node1.level-1 do
D17         repeat
D18             node2:=node1.next[i];
D19         until IS_MARKED(node2) or CAS(&node1.next[i],
             node2,GET_MARKED(node2));
D20     for i:=node1.level-1 to 0 step -1 do
D21         prev:=savedNodes[i];
D22         while true do
D23             if node1.next[i]=1 then break;
D24             last:=ScanKey(&prev,i,node1.key);
D25             RELEASE_NODE(last);
D26             if last≠node1 or node1.next[i]=1 then break;
D27             if CAS(&prev.next[i],node1,
                 GET_UNMARKED(node1.next[i])) then
```

Figure 2: The algorithm, part 1(2).

```
D28             node1.next[i]:=1;
D29             break;
D30         if node1.next[i]=1 then break;
D31         Back-Off
D32     RELEASE_NODE(prev);
D33   for i:=node1.level to maxLevel-1 do
D34       RELEASE_NODE(savedNodes[i]);
D35   RELEASE_NODE(node1);
D36   RELEASE_NODE(node1);
D37   return value;

function FindValue(value: pointer to word):integer
      return FDValue(value,false);
function DeleteValue(value: pointer to word):integer
      return FDValue(value,true);
function FDValue(value: pointer to word, delete: boolean):integer
V1    jump:=16;
V2    last:=COPY_NODE(head);
    next_jump:
V3    node1:=last;
V4    key1:=node1.key;
V5    step:=0;
V6    while true do
V7        ok=false;
V8        version:=node1.version;
V9        node2:=node1.next[0];
V10       if not IS_MARKED(node2) and node2≠NULL then
V11           version2:=node2.version;
V12           key2:=node2.key;
V13           if node1.key=key1 and node1.validLevel¿0 and
               node1.next[0]=node2 and node1.version=version
               and node2.key=key2 and node2.validLevel¿0 and
                node2.version=version2 then ok:=true;
V14       if not ok then
V15           node1:=node2:=ReadNext(&last,0);
V16           key1:=key2:=node2.key;
V17           version2:=node2.version;
V18           RELEASE_NODE(last);
V19           last:=node2;
V20           step:=0;
V21       if node2=tail then
V22           RELEASE_NODE(last);
V23           return ⊥;
V24       if node2.value=value then
V25           if node2.version=version2 then
V26               if not delete or Delete(key2,true,value)=value then
V27                   RELEASE_NODE(last);
V28                   return key2;
V29       else if ++step≥jump then
V30           COPY_NODE(node2);
V31           if node2.validLevel=0 or node2.key≠key2 then
V32               RELEASE_NODE(node2);
V33               node2:=ReadNext(&last,0);
V34               if jump≥4 then jump:=jump/2;
V35           else jump:=jump+jump/2;
V36           RELEASE_NODE(last);
V37           last:=node2;
V38           goto next_jump;
V39       else
V40           key1:=key2;
V41           node1:=node2;

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 ≥ prev.validLevel then
H7        prev:=COPY_NODE(head);
H8    else COPY_NODE(prev);
H9    while true do
H10       if node.next[level]=1 then break;
H11       for i:=prev.validLevel-1 to level step -1 do
H12           node1:=SearchLevel(&prev,i,node.key);
H13           RELEASE_NODE(prev);
H14           prev:=node1;
H15       last:=ScanKey(&prev,level,node.key);
H16       RELEASE_NODE(last);
H17       if last≠node or node.next[level]=1 then break;
H18       if CAS(&prev.next[level],node,
           GET_UNMARKED(node.next[level])) then
H19           node.next[level]:=1;
H20           break;
H21       if node.next[level]=1 then break;
H22       Back-Off
H23   RELEASE_NODE(node);
H24   return prev;
```

Figure 3: The algorithm, part 2(2).

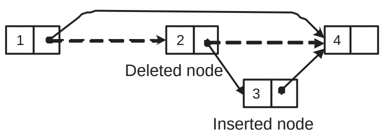

Figure 4: Concurrent insert and delete operation can delete both nodes.

have selected the lock-free memory management scheme invented by Valois [17] and corrected by Michael and Scott [8], 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 partially 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 [17] between each two normal nodes, and the latest method introduced by Harris [3] 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.

Another 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 MALLOC_NODE():pointer to Node
function READ_NODE(address:pointer to pointer to Node)
:pointer to Node
function COPY_NODE(node:pointer to Node):pointer to Node
procedure RELEASE_NODE(node:pointer to Node)
```

The function *MALLOC_NODE* allocates a new node from the memory pool of pre-allocated nodes and *RELEASE_NODE* decrements the reference counter on the corresponding given node. If the reference count reaches zero, then it calls the *ReleaseReferences* function that will call *RELEASE_NODE* on the nodes that this node has owned pointers to, and then it reclaims the node. The function *COPY_NODE* increases the reference counter for the corresponding given node and *READ_NODE* de-reference the given pointer and increase the reference counter for the corresponding node. In case the de-referenced pointer is marked, the function returns NULL.

3.2 Traversing

The functions for traversing the nodes are defined as follows:

```
function ReadNext(node1:pointer to pointer to Node
,level:integer):pointer to Node
function ScanKey(node1:pointer to pointer to Node
,level:integer,key:integer):pointer to 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*, traverses to the next node on the given level while helping any deleted nodes in between to finish the deletion. The function *ScanKey*, traverses in several steps through the next pointers at the current level until it finds a node that has the same or higher key than the given key. The argument node1 in the *ReadNext* and *ScanKey* functions are continuously updated to point to the previous node of the returned node. The full version of this paper [15] contains a more complete description of the *ReadNext* and *ScanKey* functions.

However, the use of the safe *ReadNext* and *ScanKey* operations for traversing the Skiplist, will cause the performance to be significantly lower compared to the sequential case where the next pointers are used directly. As the nodes, which are used in the lock-free memory management scheme, will be reused for the same purpose when re-allocated again after being reclaimed, the individual fields of the nodes that are not part of the memory management scheme will be intact. The *validLevel* field can therefore be used for indicating if the current node can be used for possibly traversing further on a certain level. A value of 0 indicates that this node can not be used for traversing at all, as it is possibly reclaimed or not yet inserted. As the *validLevel* field is only set to 0 directly before reclamation in line R1, a positive value indicates that the node is allocated. A value of $n + 1$ indicated that this node has been inserted up to level n. However, the next pointer of level n on the node may have been marked and thus indicating possible deletion at that level of the node. As the node is not reclaimed the *key* field is intact, and therefore it is possible to traverse from the previous node to the current position. By increasing the reference count of the node before checking the *validLevel* field, it can be assured that the node stays allocated if it was allocated directly after the incrément. Because the next pointers are always updated to point (regardless of the mark) either to nothing (NULL) or to a node that is part of the memory management, allocated or reclaimed, it is possible in some scenarios to traverse directly through the next pointers. This approach is taken by the *SearchLevel* function, see Figure 2, which traverses rapidly from an allocated node *last* and returns the node which *key* field is the highest key that is lower than the searched key at the current level. During the rapid traversal it is checked that the current key is within the search boundaries in line S23 and S11, otherwise the traversal restarts from the *last* node as this indicates that a node has been reclaimed and re-allocated while traversed. When the node suitable for returning has been reached, it is checked that it is allocated in line S11 and also assured that it then stays allocated in line S10. If this succeeds the node is returned, otherwise the traversal restarts at node *last*. If this fails twice, the traversal are done using the safe *ScanKey* operations in lines S12 to S16, as this indicates that the node possibly is inserted at the current level, but the *validLevel* field has not yet been updated. In case the node *last* is marked for deletion, it might have been deleted at the current level and thus it can not be used for traversal. Therefore the node *last* is checked if it is marked in lines S6, S20 and S26. If marked, the node *last* will be helped to fully delete on the current level and *last* is set to the previous node.

3.3 Inserting and Deleting Nodes

The implementation of the *Insert* operation, see Figure 2, starts in lines I4-I10 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 (*node*1). 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 key as of the new node, this is checked in lines I12-I23, the value of the old node (*node*2) is changed atomically with a CAS. Otherwise, in lines I24-I45 it starts trying to insert the new node starting with the lowest level increasing up to the level of the new node. The next pointers of the (to be previous) nodes 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 *Delete* operation before it has been inserted at all levels, and this is checked in lines I38 and I46. The *FindKey* operation, see Figure 2, basically follows the *Insert* operation.

The *Delete* operation, see Figure 2, starts in lines D1-D4 with a search phase to find the first node which key is equal or higher than the searched key. 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 (*node*1). When going down one level, the last node traversed on that level is remembered (*savedNodes*) for later use (this is the previous node at which the next pointer should be changed in order to delete the targeted node at that level). If the found node is the correct node, it tries to set the deletion mark of the *value* field in line D8 using the CAS primitive, and if it succeeds it also writes a valid pointer (which corresponding node will stay allocated until this node gets reclaimed) to the *prev* field of the node in line D9. This *prev* field is necessary in order to increase the performance of concurrent *HelpDelete* operations, these 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 D16-D19. Afterwards in lines D20-D32 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 operations that are in the search phase 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 *Delete* 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 3, 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 starts the search at the head node. In lines H11-H16 it searches for the correct node (*prev*). The actual deletion of this node on the current level takes place in line H18. Lines H10-H22 will be repeated until the node is deleted at the current level. This operation might execute concurrently with the corresponding *Delete* operation as well with other *HelpDelete* operations, and therefore all operations synchronize with each other in lines D23, D26, D28, D30, H10, H17, H19 and H21 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 *Delete* operations that stops the progress of the current operation, puts the current 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.

3.4 Value Oriented Operations

The *FindValue* and *DeleteValue* operations, see Figure 3, traverse from the head node along the lowest level in the Skiplist until a node with the searched value is found. In every traversal step, it has to be assured that the step is taken from a valid node to a valid node, both valid at the same time. The *validLevel* field of the node can be used to safely verify the validity, unless the node has been reclaimed. The *version* field is incremented by the *Insert* operation in line I30, after the node has been inserted at the lowest level, and directly before the *validLevel* is set to indicate validity. By performing two consecutive reads of the *version* field with the same contents, and successfully verifying the validity in between the reads, it can be concluded that the node has stayed valid from the first read of the version until the successful validity check. This is done is lines V8-V13. If this fails, it restarts and traverses the safe node *last* one step using the *ReadNext* function in lines V14-V21. After a certain number (*jump*) of successful fast steps, an attempt to advance the *last* node to the current position is performed in lines V29-V38. If this attempt succeeds, the threshold *jump* is increased by 1 1/2 times, otherwise it is halved. The traversal is continued until a node with the searched value is reached in line V24 or that the tail node is reached in line V21. In case the found node should be deleted, the *Delete* operation is called for this purpose in line V26.

4. CORRECTNESS

We have defined a well precised semantics of the operations on our dictionary implementation, described in the full version of this paper [15]. We have proved that our algorithm is linearizable [5] and that it is lock-free. Because of space constraints we have moved the full details of the proofs to the full version of the paper, and only present a brief description of the linearizability points [5]:

An *Insert* operation which succeeds takes effect atomically at the CAS sub-operation in line I26.

An *Insert* operation which updates takes effect atomically at the CAS sub-operation in line I13.

A *FindKey* operation which succeeds takes effect atomically at the read sub-operation of the *value* field in line F8.

A *FindKey* operation which fails takes effect atomically at either i) the read sub-operation of *READ_NODE* in line N2 or N5 (from K1 or K5, from F6), ii) the read sub-operation of the *value* field in line F8.

A *DeleteKey* operation which succeeds takes effect atomically at the CAS sub-operation in line D8.

A *DeleteKey* operations which fails takes effect atomically at either i) the read sub-operation of *READ_NODE* in line N2 or N5 (from K1 or K5, from D4), ii) the read sub-operation of the *value* field in line D6.

A *FindValue* operation which succeed takes effect atomically at the read sub-operation of the *value* field in line V24.

A *FindValue* operation which fails takes effect atomically at either i) the hidden read sub-operation of the next pointer of node *node*1 in the *READ_NODE* function in line N2 or N5 (from V15), ii) the read sub-operation of the *value* field in line V24, iii) the read sub-operation of the *head* node in line V2, iv) the concurrent successful CAS sub-operation on marking the *value* field in line D8 that can be ordered before the read sub-operation of the same *value* field in line V24, and after the read sub-operation of the *head* node in line V2.

A *DeleteValue* operation which succeeds takes effect atomically at the CAS sub-operation in line D8.

A *DeleteValue* operation which fails takes effect atomically at the same statement as the *FindValue* operation.

5. EXPERIMENTS

We have performed experiments on both the limited set of operations on a dictionary (i.e. the *Insert*, *FindKey* and *DeleteKey* operations), as well as on the full set of operations on a dictionary (i.e. also including the *FindValue* and *DeleteValue* operations).

In our experiments with the limited set of operations on a dictionary, each concurrent thread performed 20000 sequential operations, whereof the first 50 up to 10000 of the totally performed operations are *Insert* operations, and the remaining operations was randomly chosen with a distribution of 1/3 *Insert* operations versus 1/3 *FindKey* and 1/3 *DeleteKey* operations. For the systems which also involve preemption, a synchronization barrier was performed between the initial insertion phase and the remaining operations. The key values of the inserted nodes was randomly chosen between 0 and $1000000 * n$, where n is the number of threads. Each experiment was repeated 50 times, and an average execution time for each experiment was estimated. Exactly the same sequential operations were performed for all different implementations compared. Besides our implementation, we also performed the same experiment with the lock-free implementation by Michael [6] which is the most recently claimed to be one of the most efficient concurrent dictionaries existing.

Our experiments with the full set of operations on a dictionary, was performed similarly to the experiments with the limited set of operations, except that the remaining operations after the insertion phase was randomly chosen with a distribution of 1/3 *Insert* operations versus 15/48 *FindKey*, 15/48 *DeleteKey*, 1/48 *FindValue* and 1/48 *DeleteValue* operations. Each experiment was repeated 10 times. Besides our implementation, we also performed the same experiment with a lock-based implementation of Skiplists using a single global lock.

The Skiplist-based implementations have a fixed level of 10, which corresponds to an expected optimal performance with an average of 1024 nodes. All lock-based implementations are based on simple spin-locks using the TAS atomic primitive. A clean-cache operation was performed just before each sub-experiment using a different implementation. 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 450 MHz Pentium II PC running Linux. In order to evaluate our algorithm with full concurrency we also used a SGI Origin 2000 system running Irix 6.5 with 64 195 MHz MIPS R10000 processors. The results from the experiment with a limited set of operations on the SGI and the Linux systems are shown in Figure 5. The results of the additional experiments are available in the full version of this paper [15]. The average execution time is drawn as a function of the number of threads. Observe that the scale is different on each figure in order to clarify the experiments on the individual implementations as much as possible. For the SGI system and the limited set of operations, our lock-free algorithm shows a negative time complexity with respect to the size, though for the full set of operations the performance conforms to be averagely the same independently of the size. Our conjecture for this behavior is that the performance of the ccNUMA memory model of the SGI system increases significantly when the algorithm works on disjoint parts of the memory (as will occur with large sizes of the dictionary), while the time spent by the search phase of the operation will vary insignificantly because of the expected logarithmic time complexity. On the other hand, for the full set of operations, there will be corresponding performance degradation because of the linear time complexity for the value oriented operations. However, for the algorithm by Michael [6] the benefit for having disjoint access to the memory is insignificant compared to the performance degradation caused by the linear time complexity.

Our lock-free implementation scales best compared to the other implementation, having best performance for realistic sizes and any number of threads, i.e. for sizes larger or equal than 500 nodes, independently if the system is fully concurrent or involves a high degree of pre-emptions. On scenarios with the full set of operations our algorithm performs better than the simple lock-based Skiplist for more than 3 threads on any system.

6. CONCLUSIONS

We have presented a lock-free algorithmic implementation of a concurrent dictionary. 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 dictionaries our algorithm is lock-free and avoids the performance penalties that come with the use of locks. Compared to the previous non-blocking concurrent dictionary algorithms, our algorithm inherits and carefully retains the basic design characteristic that makes Skiplists practical: logarithmic search time complexity. Previous non-blocking algorithms did not perform well on dictionaries with realistic sizes because of their linear or worse search time complexity. Our algorithm also implements the full set of operations that is needed in a practical setting.

An interesting future work would be to investigate if it is suitable and how to change the Skiplist level reactively to the current average number of nodes. Another issue is how to choose and change the lengths of the fast jumps in order to get maximum performance of the *FindValue* and *DeleteValue* operations.

We compared our algorithm with the most efficient non-blocking implementation of dictionaries known. Experiments show that our implementation scales well, and for realistic number of nodes our implementation outperforms the other implementation, 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.

7. REFERENCES

[1] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, 2000.

[2] L. Boug, J. Gabarr, and X. Messeguer. Concurrent AVL revisited: Self-balancing distributed search trees. Research Report RR95-45, LIP, ENS Lyon, 1995.

[3] T. L. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th*

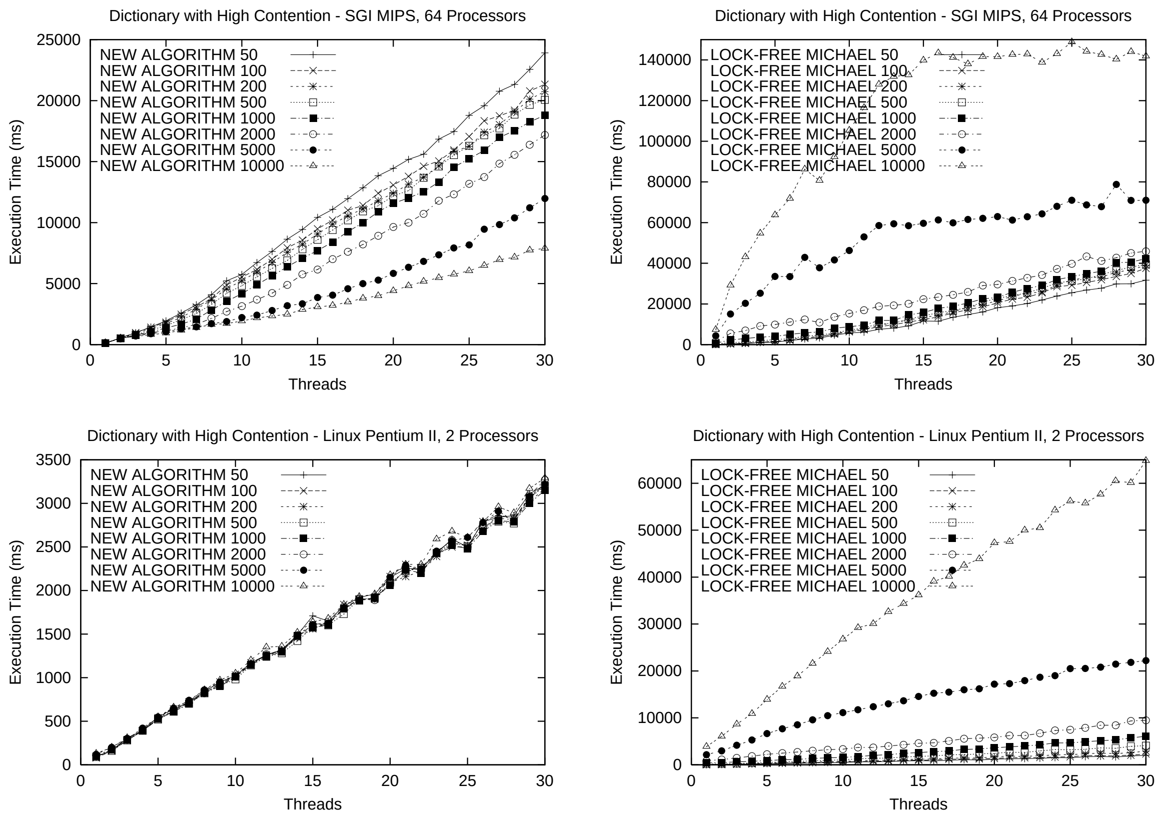

Figure 5: Experiment with dictionaries and high contention, initialized with 50,100,...,10000 nodes

International Symposium of Distributed Computing, pages 300–314, Oct. 2001.

[4] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, Jan. 1991.

[5] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[6] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, 2002.

[7] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 21–30, 2002.

[8] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical report, Computer Science Department, University of Rochester, 1995.

[9] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[10] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123, 1990.

[11] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.

[12] A. Silberschatz and P. Galvin. *Operating System Concepts*. Addison Wesley, 1994.

[13] H. Sundell and P. Tsigas. NOBLE: A non-blocking inter-process communication library. In *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, Lecture Notes in Computer Science. Springer Verlag, 2002.

[14] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*. IEEE press, 2003.

[15] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries, extended version. Technical report, Computing Science, Chalmers University of Technology, Dec. 2003.

[16] P. Tsigas and Y. Zhang. Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In *Proceedings of the 3rd ACM Workshop on Software and Performance*, pages 55–67. ACM Press, 2002.

[17] J. D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, 1995.