Technical Report no. 2003-10

Scalable and Lock-Free Concurrent Dictionaries[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. 2003-10
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 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 to use 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. Using 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.

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 [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 steps, regardless of the actions of the concurrent operations. Non-blocking algorithms have been shown to be of big practical importance [15, 16], 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 majority 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 small 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 search complexity 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 using concurrent ordered lists 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

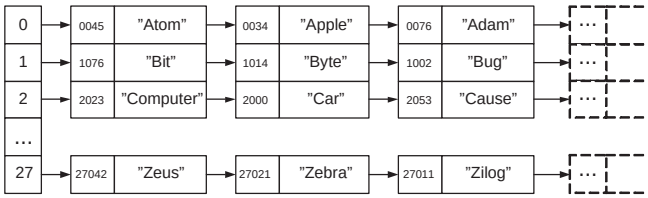

Figure 1. Example of a Hash Table with Dictionaries as branches.

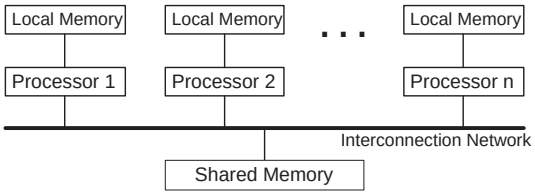

Figure 2. Shared Memory Multiprocessor System Structure

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, as in Figure 1. 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 three 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 define the properties of the system 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.

2 System Description

A typical abstraction of a shared memory multiprocessor system configuration is depicted in Figure 2. 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, see Figure 3. 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 6 as it is used in this implementation. In all code figures in this section, arrays are indexed starting from 0.

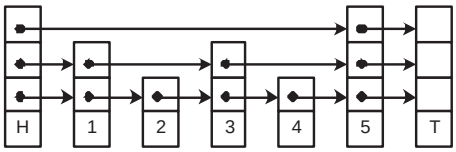

Figure 3. The Skiplist data structure with 5 nodes inserted.

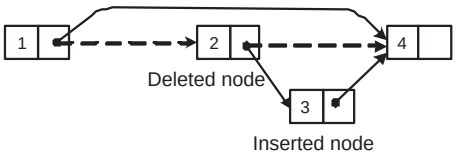

Figure 4. Concurrent insert and delete operation can delete both nodes.

```
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.

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.

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 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
```

```
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 ReadNext(node1:pointer to pointer to Node, level:integer)
 :pointer to Node
N1    if IS_MARKED((*node1).value) then
        *node1:=HelpDelete(*node1,level);
N2    node2:=READ_NODE((*node1).next[level]);
N3    while node2=NULL do
N4       *node1:=HelpDelete(*node1,level);
N5       node2:=READ_NODE((*node1).next[level]);
N6    return node2;

function ScanKey(node1:pointer to pointer to Node, level:integer
 , key:integer):pointer to Node
K1    node2:=ReadNext(node1,level);
K2    while node2.key < key do
K3       RELEASE_NODE(*node1);
K4       *node1:=node2;
K5       node2:=ReadNext(node1,level);
K6    return node2;

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 true2;
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;
```

Figure 6. The algorithm, part 1(3).

```
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
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;
```

Figure 7. The algorithm, part 2(3).

```
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 8. The algorithm, part 3(3).

```
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*, see Figure 6, traverses to the next node on the given level while helping any deleted nodes in between to finish the deletion. The function *ScanKey*, see Figure 6, 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.

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 reallocated 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 increment. 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 6, 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 6, 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 ($node1$). 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 key as of the new node, this is checked in lines I12-I23, the value of the old node ($node2$) 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 6, basically follows the *Insert* operation.

The *Delete* operation, see Figure 6, 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 ($node1$). 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.

3.4 Helping Scheme

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 8, tries to fulfill the deletion on the current level and returns when it is completed. It starts in lines H1-H4 with setting the dele-

tion 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.

3.5 Value Oriented Operations

The *FindValue* and *DeleteValue* operations, see Figure 7, 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

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 dictionary at the time t. L_t is viewed as a set of unique pairs $\langle k, v\rangle$ consisting of a unique key k and a corresponding unique value v. The operations that can be performed on the dictionary are $Insert$ (I), $FindKey$ (FK), $DeleteKey$ (DK), $FindValue$ (FV) and $DeleteValue$ (DV). 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:*

$$\langle k_1, \_\rangle \notin L_{t_1} : \mathbf{I_1}(\langle \mathbf{k_1}, \mathbf{v_1}\rangle) = \mathbf{true}, \quad \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \cup \{\langle \mathbf{k_1}, \mathbf{v_1}\rangle\} \tag{1}$$

$$\langle k_1, v_{1_1}\rangle \in L_{t_1} : \mathbf{I_1}(\langle \mathbf{k_1}, \mathbf{v_{1_2}}\rangle) = \mathbf{true_2}, \quad \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \setminus \{\langle \mathbf{k_1}, \mathbf{v_{1_1}}\rangle\} \cup \{\langle \mathbf{k_1}, \mathbf{v_{1_2}}\rangle\} \tag{2}$$

$$\langle k_1, v_1\rangle \in L_{t_1} : \mathbf{FK_1}(\mathbf{k_1}) = \mathbf{v_1} \tag{3}$$

$$\langle k_1, v_1\rangle \notin L_{t_1} : \mathbf{FK_1}(\mathbf{k_1}) = \perp \tag{4}$$

$$\langle k_1, v_1\rangle \in L_{t_1} : \mathbf{DK_1}(\mathbf{k_1}) = \mathbf{v_1}, \quad \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \setminus \{\langle \mathbf{k_1}, \mathbf{v_1}\rangle\} \tag{5}$$

$$\langle k_1, v_1\rangle \notin L_{t_1} : \mathbf{DK_1}(\mathbf{k_1}) = \perp \tag{6}$$

$$\langle k_1, v_1\rangle \in L_{t_1} : \mathbf{FV_1}(\mathbf{v_1}) = \mathbf{k_1} \tag{7}$$

$$\langle k_1, v_1\rangle \notin L_{t_1} : \mathbf{FV_1}(\mathbf{v_1}) = \perp \tag{8}$$

$$\langle k_1, v_1\rangle \in L_{t_1} : \mathbf{DV_1}(\mathbf{v_1}) = \mathbf{k_1}, \quad \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \setminus \{\langle \mathbf{k_1}, \mathbf{v_1}\rangle\} \tag{9}$$

$$\langle k_1, v_1\rangle \notin L_{t_1} : \mathbf{DV_1}(\mathbf{v_1}) = \perp \tag{10}$$

Note that the operations will work correctly also if relaxing the condition that values are unique. However, the results of the $FindValue$ and $DeleteValue$ operations will be undeterministic in the sense that it is not decidable which key value that will be returned in the presence of several key-value pairs with the same value. In the case of the $DeleteValue$ operation, still only one pair will be removed.

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 $\parallel$ 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 $\langle k, v\rangle$ is present ($\langle k, v\rangle \in L$) 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 $\langle k, v\rangle$, 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 dictionary.*

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. □

As we are using a lock-free memory management scheme with a fixed memory size and where reclaimed nodes can only be allocated again for the same purpose, we know that there is a fixed number of nodes that will be used with the Skiplist, and that the individual fields (like key, value, next etc.) of the nodes are only changed by the operations of this algorithm.

Definition 5 *A node that is used in the Skiplist is defined as **valid** if the node is inserted at the lowest level, i.e. there is a next pointer on any other valid node that points (disregarding the eventual mark) to this node, or the node the $validLevel$ field set to higher than zero and has been fully deleted but not yet been reclaimed. All other nodes are defined as **invalid**, i.e. the node is reclaimed, or has been allocated but not yet inserted at the lowest level. A node that is used in the Skiplist is defined as **valid at level i** if the node is inserted at level i, i.e. there is a next pointer at level i on any other valid node that points (disregarding the eventual mark) to this node, or the node has the $validLevel$ field set higher than i and has been fully deleted but not yet been reclaimed.*

An interesting observation of the previous definition is that if a node is present in the abstract internal state L then it is also valid, and that if a node is valid then also the individual fields of the node are valid.

Lemma 2 *A valid node with a increased reference count, can always be used to continue traversing from, even if the node is deleted.*

Proof: For every instruction in the algorithm that increments the reference count (i.e. the $READ\_NODE$ and $COPY\_NODE$ functions), there exists a corresponding instruction that decrements the reference count equally much (i.e. the $RELEASE\_NODE$ function). This means that if the reference count has been incremented, that the reference count can not reach zero (and thus be reclaimed) until the corresponding decrement instruction is executed. A node with a increased reference count can thus not be reclaimed, unless it already was reclaimed before the reference count was incremented. As the node is valid, the key field is also valid, which means that we know the absolute position in the Skiplist. If the node is not deleted the next pointers can be used for traversing. Otherwise it is always possible to get to the previous node by searching from the head of the Skiplist using the key, and traverse from there. □

Lemma 3 *The node $node1$ that is found in line S17 of the $SearchLevel$ function, is a valid node with a increased reference count and will therefore not be reclaimed by concurrent tasks, and is (or was) the node with the nearest key that is lower than the searched key.*

Proof: The reference count is incremented in line S10 before the check for validity in line S11, which means that if the validity test succeeds then the node will stay valid. The $validLevel$ field of a node is set in lines I31 and I42 to the current level plus one after each successful step of the $Insert$ operation, and is set to zero in line R1 just before the node is fully deleted and will be reclaimed. This means that if the $validLevel$ field is more than the current level, that the node is valid. Alternatively if the node is the same as a known valid node $last$, then it is also valid. If the node is valid, it is also checked that the key value is lower than was searched for. Before the validity check, the next node of $node1$ was read as $node2$ in line S4 and its key was checked to be more than or equal to the searched key in line S9. This means that the node $node1$ in line S17 is valid and was (or still is) the node with the nearest key that is lower than the searched key. □

Lemma 4 *The node $node2$ that is found in line V26 of the $FindValue$ and $DeleteValue$ functions, was present during the read of its value and this value was the same as searched for.*

Proof: The $Delete$ operation marks the value before it starts with marking the next pointers and removing the node from the Skiplist. A node that is valid and the value is non-marked, is therefore present in the dictionary as the node must be inserted at the lowest level and not yet deleted. The node was valid in line V13 as the $validLevel$ field was positive. The $version$ field is only incremented in line I30, directly before the $validLevel$ field becomes positive. As the version was the same before the check for validity in line V13 as well as after the check for equalness and validity in V24, this means that the node has been valid all the time. □

Lemma 5 *The node $node2$ that is found in line V37 of the $FindValue$ and $DeleteValue$ functions, is a valid node at the current, or between the previously known safe node $last$ and the current position along the searchpath. The node also has a increased reference count and will therefore not be reclaimed by concurrent tasks.*

Proof: The reference count is incremented in line V30 before the check for validity in line V31, which means that if the validity test succeeds then the node will stay valid. The validity check follows Lemma 3. If the node was not valid or the key of the node did not match the current position in the searchpath (i.e. the key field has changed due to reclamation of the node before the increment of the reference count) then $node2$ will be set to the next node of $last$ using the $ReadNext$ function. □

Lemma 6 *The functions $FindValue$ and $DeleteValue$ will not skip any nodes while traversing the Skiplist from left to right, and will therefore traverse through all nodes that was present in the Skiplist at the start of the traversing and which was still present while traversed.*

Proof: In order to safely move from $node1$ to $node2$, it has to be assured that both nodes are valid and that $node1$ has been pointing to $node2$ at the lowest level while both were valid, and that $node1$ is at the current position (i.e. the key field). If this holds we can conclude that $node2$ is, or was at the time of starting the traversal, the very next node of $node1$. These properties are checked in line V13, where the validity is confirmed to have hold during the check of the other properties by that the $version$ fields of both nodes were the same as in lines V8 and V11 before checking the validity using the $validLevel$ field. If the check in line V13 failed, then $node2$ will be set to the next node of $last$ using the $ReadNext$ function, which position is between the previously known safe node $last$ and the current position along the searchpath. Given by Lemma 5, when the node $last$ is updated, it is always set to a valid node with a position between the next node of the previously known safe node $last$ and the current position along the searchpath. Consequently, the functions $FindValue$ and $DeleteValue$ will not skip any nodes while traversing the Skiplist from left to right. □

Definition 6 *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 after 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 the state of the dictionary, which result the decision point depends on is read. We also define the state-change point as the atomic statement where the operation changes the abstract internal state of the dictionary after it has passed the corresponding decision point.*

We will now use these points in order to show the existance and location in execution history of a point where the concurrent operation can be viewed as it occured atomically, i.e. the *linearizability point*.

Lemma 7 *An $Insert$ operation which succeeds ($I(\langle k, v\rangle) = true$), takes effect atomically at one statement.*

Proof: The decision point for an $Insert$ operation which succeeds ($I(\langle k, v\rangle) = true$), is when the CAS sub-operation in line I26 (see Figure 6) 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 $\langle k, \_\rangle \notin L_{t_1}$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle k, v\rangle \in L_{t_2}$. Consequently, the linearizability point will be the CAS sub-operation in line I26. □

Lemma 8 *An $Insert$ operation which updates ($I(\langle k, v\rangle) = true_2$), takes effect atomically at one statement.*

Proof: The decision point for an $Insert$ operation which updates ($I(\langle k, v\rangle) = true_2$), is when the CAS will succeed in line I13. The state of the list (L_{t_1}) directly before passing the decision point must have been $\langle k, \_\rangle \in L_{t_1}$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle k, v\rangle \in L_{t_3}$. Consequently, the linearizability point will be the CAS sub-operation in line I13. □

Lemma 9 *A $FindKey$ operation which succeeds ($FK(k) = v$), takes effect atomically at one statement.*

Proof: The decision point for a $FindKey$ operation which succeeds ($FK(k) = v$), is when the check for marked value in line F9 fails. The state-read point is when the value of the node is read in line F8. As the key field of the node can not change concurrently, the state of the list (L_{t_1}) directly before passing the state-read point must have been $\langle k, v\rangle \in L_{t_1}$. Consequently, the linearizability point will be the read sub-operation of the $value$ field in line F8. □

Lemma 10 *A $FindKey$ operation which fails ($FK(k) = \perp$), takes effect atomically at one statement.*

Proof: The decision point for a $FindKey$ operation which fails ($FK(k) = \perp$), is when the check for key equality fails or when the check for marked value in line F9 succeeds. If the key equality in line F9 fails, the state-read point is the read sub-operation of $READ\_NODE$ in line N2 or N5 (from K1 or K5, from F6) when the next pointer at lowest level of the previous node is read. If the check for marked value in line F9 succeeds, the state-read point is the read sub-operation of the $value$ field in line F8. In both cases, the state of the list (L_{t_1}) directly before passing the state-read point must have been $\langle k, v\rangle \notin L_{t_1}$. Consequently, the linearizability point will be either of the state-read points. □

Lemma 11 *A $DeleteKey$ operation which succeeds ($DK(k) = v$), takes effect atomically at one statement.*

Proof: The decision point for a $DeleteKey$ operation which succeeds ($DK(k) = v$) is when the CAS sub-operation in line D8 (see Figure 7) succeeds. The state of the list (L_t) directly before passing of the decision point must have been $\langle k, v\rangle \in L_t$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle k, v\rangle \notin L_t$. Consequently, the linearizability point will be the CAS sub-operation in line D8. □

Lemma 12 *A $DeleteKey$ operations which fails ($DK(k) = \perp$), takes effect atomically at one statement.*

Proof: The decision point for a $DeleteKey$ operation which fails ($DK(k) = \perp$), is when the check for key equality fails or when the check for non-marked value in line D7 fails. If the key equality in line D7 fails, the state-read point is the read sub-operation of $READ\_NODE$ in line N2 or N5 (from K1 or K5, from D4) when the next pointer at lowest level of the previous node is read. If the check for non-marked value in line D7 fails, the state-read point is the read sub-operation of the $value$ field in line D6. In both cases, the state of the list (L_{t_1}) directly before passing the state-read point must have been $\langle k, v\rangle \notin L_{t_1}$. Consequently, the linearizability point will be either of the state-read points. □

Lemma 13 *A $FindValue$ operation which succeeds ($FV(v) = k$), takes effect atomically at one statement.*

Proof: The decision point for a $FindValue$ operation which succeeds ($FV(v) = k$), is when the check for valid node (and also a valid $value$ field in line V24) in line V25 succeeds. The state-read point is when the $value$ field is read in line V24. As the key field of the node can not change concurrently and as given by Lemma 4, the state of the list (L_{t_1}) directly before passing the state-read point must have been $\langle k, v\rangle \in L_{t_1}$. Consequently, the linearizability point will be the read sub-operation of the $value$ field in line V24. □

Lemma 14 *A $FindValue$ operation which fails ($FV(v) = \perp$), takes effect atomically at one statement.*

Proof: For a $FindValue$ operation which fails ($FV(v) = \perp$), all checks for value equality in line V24 fails. Because of the uniqueness of values, there can be at most one pair $\langle k_1, v_1\rangle$ present in the dictionary at one certain moment of time where $v = v_1$. Given by Lemma 6 we know that the algorithm will pass by the node with key k_1 if $\langle k_1, \_\rangle \in L_{t_1}$ at the time of traversal, and that all keys in the possible range of keys will be passed by as we start traversing from the lowest key and that the Skiplist is ordered.

If during the execution, $key1 < k_1 < key2$, then if the check in line V13 succeeds, the state-read point is the read sub-operation in line V9, otherwise if the check in line V13 fails, the state-read point is the hidden read sub-operation

of the next pointer of node $node1$ in the $READ\_NODE$ function in line N2 or N5 (from V15). The state of the list (L_{t_1}) directly before passing the state-read point must have been $\langle k_1, v_1\rangle \notin L_{t_1}$. Consequently, the linearizability point will be the state-read point.

If during the execution, $key2 = k_1$ and the $value$ field of node $node2$ was not equal to v in line V24, then the state-read point will be the read sub-operation of the $value$ field in line V24. The state of the list (L_{t_1}) directly before passing the state-read point must have been $\langle k_1, v_1\rangle \notin L_{t_1}$. Consequently, the linearizability point will be the state-read point.

As all operations on shared memory as read, write and atomic primitives, are atomic, they can be totally ordered. If during the execution, $key2 = k_1$ and the $value$ field of node $node2$ was marked in line V24, the linearizability point will be the concurrent successful CAS sub-operation on the same $value$ field in line D8 that can be ordered before the read sub-operation in line V24, and after the read sub-operation of the $head$ node in line V2. If no such concurrent CAS sub-operation exists, the linearizability point will be the read sub-operation of the $head$ node in line V24. The state of the list (L_{t_1}) directly after passing the linearizability point must have been $\langle k_1, v_1\rangle \notin L_{t_1}$. □

Lemma 15 *A $DeleteValue$ operation which succeeds ($DV(v) = k$), takes effect atomically at one statement.*

Proof: The decision point for a $DeleteValue$ operation which succeeds ($DV(v) = k$) is when the CAS sub-operation in line D8 (from V26) succeeds. The state of the list (L_t) directly before passing of the decision point must have been $\langle k, v\rangle \in L_t$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle k, v\rangle \notin L_t$. Consequently, the linearizability point will be the CAS sub-operation in line D8. □

Lemma 16 *A $DeleteValue$ operation which fails ($DV(v) = \perp$), takes effect atomically at one statement.*

Proof: The proof is the same as for $FindValue$, see Lemma 14. □

Definition 7 *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, $E_1 \implies E_2$ means that if E_1 holds then E_2 holds as well, and $\oplus$ stands for exclusive or (i.e. $a \oplus b$ means $(a \vee b) \wedge \neg(a \wedge b)$):*

$$\mathbf{Op_1} \rightarrow_{\mathbf{d}} \mathbf{Op_2}, \nexists \mathbf{Op_3}.\mathbf{Op_1} \Rightarrow_{\mathbf{d}} \mathbf{Op_3}, \nexists \mathbf{Op_4}.\mathbf{Op_4} \Rightarrow_{\mathbf{d}} \mathbf{Op_2} \implies \mathbf{Op_1} \Rightarrow_{\mathbf{d}} \mathbf{Op_2} \quad (11)$$

$$\mathbf{Op_1} \parallel \mathbf{Op_2} \implies \mathbf{Op_1} \Rightarrow_{\mathbf{d}} \mathbf{Op_2} \oplus \mathbf{Op_2} \Rightarrow_{\mathbf{d}} \mathbf{Op_1} \quad (12)$$

$$\mathbf{Op_1} \Rightarrow_{\mathbf{d}} \mathbf{Op_2} \implies \mathbf{Op_1} \Rightarrow \mathbf{Op_2} \quad (13)$$

$$\mathbf{Op_1} \Rightarrow \mathbf{Op_2}, \mathbf{Op_2} \Rightarrow \mathbf{Op_3} \implies \mathbf{Op_1} \Rightarrow \mathbf{Op_3} \quad (14)$$

Lemma 17 *The operations that are directly totally ordered using formula 11, form an equivalent valid sequential execution.*

Proof: If the operations are assigned their direct total order ($Op_1 \Rightarrow_d Op_2$) by formula 11 then also the linearizability point of Op_1 is executed before the respective point 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 18 *The operations that are directly totally ordered using formula 12 can be ordered unique and consistent, and form an equivalent valid sequential execution.*

Proof: Assume we order the overlapping operations according to their linearizability points. As the state before as well as after the linearizability points is identical to the corresponding state defined in the semantics of the respective sequential operations in formulas 1 to 10, we can view the operations as occurring at the linearizability point. As the linearizability points consist of atomic operations and are therefore ordered in time, no linearizability point can occur at the very same time as any other linearizability point, therefore giving a unique and consistent ordering of the overlapping operations. □

Lemma 19 *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 loop retries 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 $Insert$, $FindKey$, $DeleteKey$, $FindValue$ or $DeleteValue$ 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 dictionary.*

Proof: Following from Lemmas 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17 and 18 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 [9], we could use the corresponding proof of termination. This together with Lemma 19 and that the state is only changed at one atomic statement (Lemmas 1,7,8,11,15), gives that our implementation is lock-free. □

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 involves preemption, a synchronization barrier was performed between the initial insertion phaes 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 preemptive 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 these experiments are shown in Figures 9 and 10 . 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 clearify 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

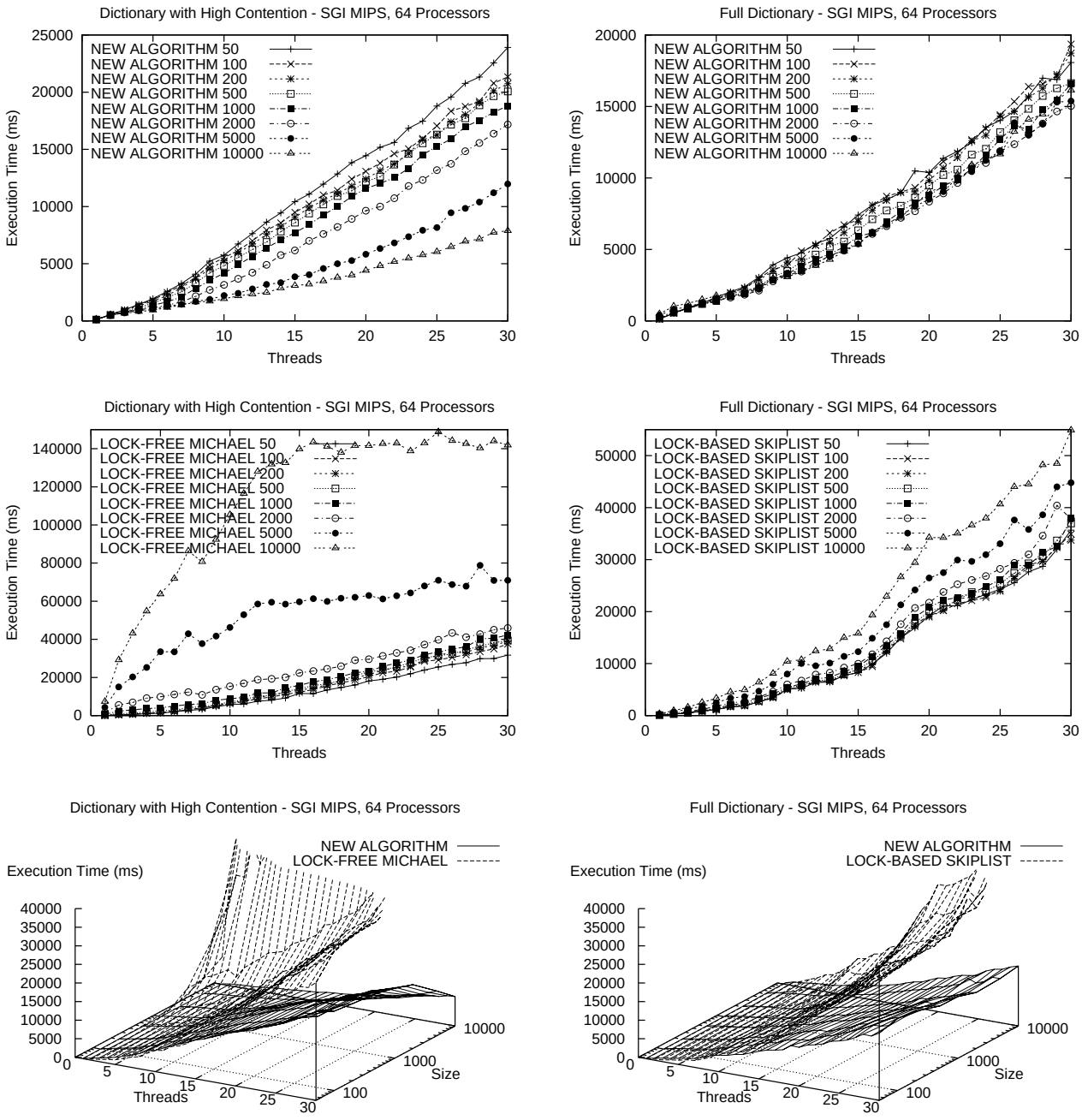

Figure 9. Experiment with dictionaries and high contention on SGI Origin 2000, initialized with 50,100,...,10000 nodes

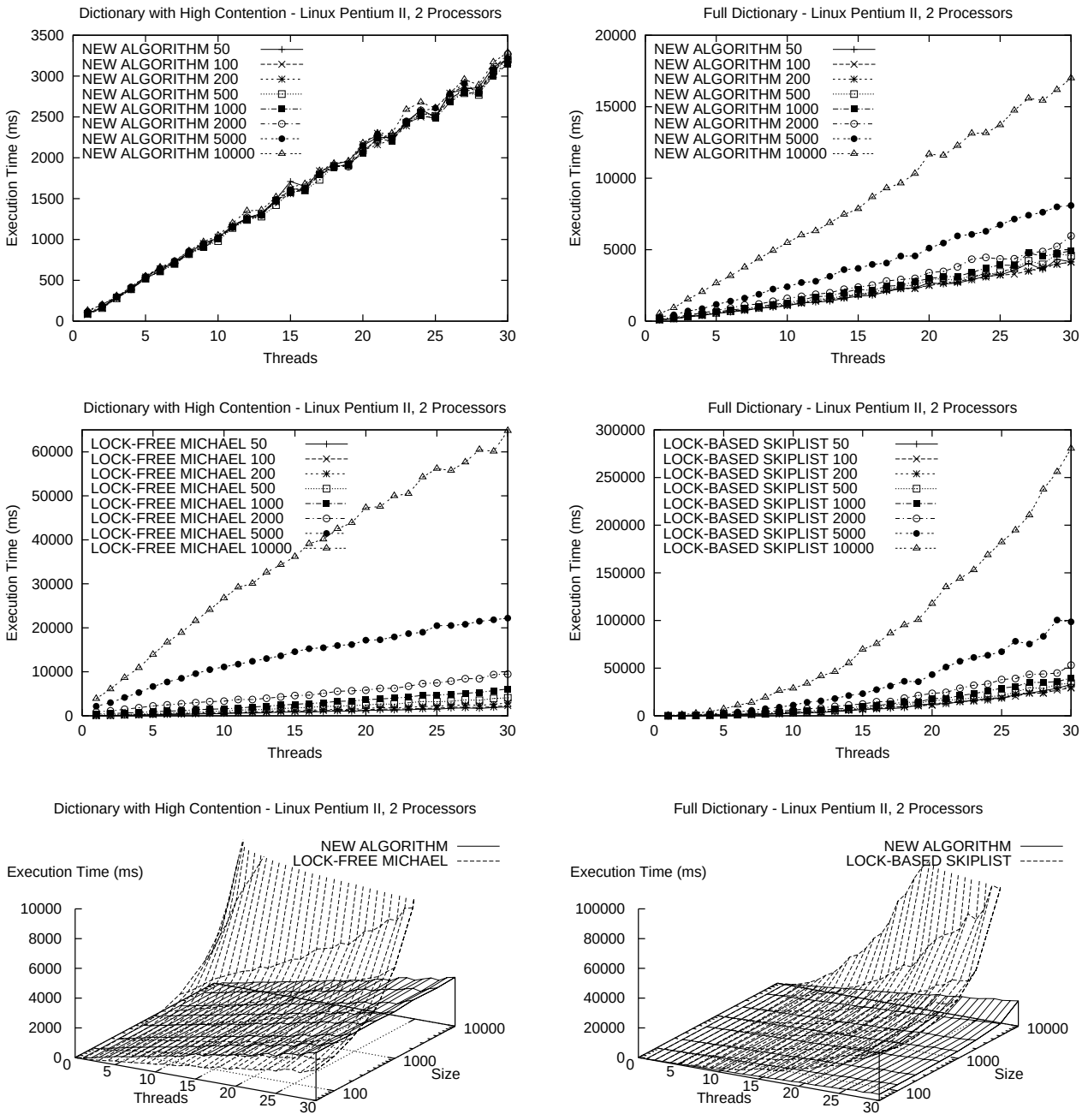

Figure 10. Experiment with dictionaries and high contention on Linux Pentium II, initialized with 50,100,...,10000 nodes

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.

References

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

[2] L. Bougé, J. Gabarró, and X. Messeguer, "Concurrent avl revisited: Self-balancing distributed search trees," LIP, ENS Lyon, Research Report RR95-45, 1995.

[3] T. L. Harris, "A pragmatic implementation of non-blocking linked lists," in *Proceedings of the 15th International Symposium of Distributed Computing*, Oct. 2001.

[4] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, pp. 124–149, Jan. 1991.

[5] M. Herlihy and J. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 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*, 2002, pp. 73–28.

[7] ——, "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*, 2002.

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

[9] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–76, 1990.

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

[11] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 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*, ser. Lecture Notes in Computer Science. Springer Verlag, 2002.

[14] ——, "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] P. Tsigas and Y. Zhang, "Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors," in *Proceedings of the international conference on Measurement and modeling of computer systems (SIGMETRICS 2001)*. ACM Press, 2001, pp. 320–321.

[16] ——, "Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies," in *Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP '02)*. ACM Press, 2002.

[17] J. D. Valois, “Lock-free data structures,” Ph.D. dissertation, Rensselaer Polytechnic Institute, Troy, New York, 1995.