Almost Wait-free Resizable Hashtables

Gao, H.
Department of Computing Science
University of Groningen
9700 AV Groningen, The Netherlands
hui@cs.rug.nl

Groote, J.F.
Department of Computing Science
Eindhoven University of Technology
5600 MB Eindhoven, The Netherlands
jfg@win.tue.nl

Hesselink, W.H.
Department of Computing Science
University of Groningen
9700 AV Groningen, The Netherlands
wim@cs.rug.nl

Abstract

In multiprogrammed systems, synchronization often turns out to be a performance bottleneck and the source of poor fault-tolerance. Wait-free and lock-free algorithms can do without locking mechanisms, and therefore do not suffer from these problems. We present an efficient almost wait-free algorithm for parallel accessible hashtables, which promises more robust performance and reliability than conventional lock-based implementations. Our solution is as efficient as sequential hashtables. It can easily be implemented using C-like languages and requires on average only constant time for insertion, deletion or accessing of elements. The algorithm allows the hashtables to grow and shrink when needed.

A true problem of wait-free and lock-free algorithms is that they are hard to design correctly, even when apparently straightforward. The reason for this is that processes can execute all statements in every conceivable order. Since our algorithm is quite large and rather complex, we turned to the interactive theorem prover PVS to prove safety of our algorithm, which we could not have done reliably by hand. To our knowledge no algorithms of comparable complexity have ever been mechanically verified. Wait-freedom is shown informally.

1. Introduction

We are interested in efficient, reliable, parallel algorithms. The classical synchronization paradigms are not most suited for this, because synchronization often turns out a performance bottleneck, and failure of a single process can force all other processes to come to a halt. Therefore, wait-free or lock-free algorithms are of interest [9, 17, 11].

An algorithm is *wait-free* when each process can accomplish its task in a finite number of steps, independently of the activity and speed of other processes. An algorithm is *lock-free* when it guarantees that within a finite number of steps always some process will complete its tasks, even if other processes halt. The difference between wait-free and lock-free is that a lock-free process can be arbitrarily delayed by other processes that repeatedly start and accomplish tasks.

Since the processes in a wait-free algorithm run rather independently of each other, wait-free algorithms scale up well when there are more processes. Processors can finish their tasks on their own, without being blocked, and generally even without being delayed by other processes. When there are processors of differing speeds, or under different loads, a wait-free algorithm will generally distribute common tasks over all processors, such that it is finished as quickly as possible. A wait-free algorithm can carry out its task even when all but one processor stops working. Without problem it can stand any pattern of processors being switched off and on again. The only noticeable effect of failing processors is that common tasks will be carried out slower and that the failing processor may have claimed resources, such as memory, that it can not relinquish anymore.

Despite their technical advantages, there are relatively few wait-free algorithms, due to the inherent complexity of these algorithms. There are very general solutions for wait-free data structures in general [1, 2, 5, 7, 8], but these are not efficient. Furthermore, there exist wait-free algorithms for a small number of domains, such as linked lists [17], queues

[18] and memory management [6, 9].

In this paper we present an almost wait-free algorithm for one of the most important data structures in current use, namely hashtables, by means of synchronization primitives *compare&swap*, *test&set* and *fetch&add*. Hashtables are used to store huge but sparsely filled tables, and they are virtually used everywhere, e.g., in compilers, run-time environments and application software. In our algorithm processes can simultaneously insert, delete and find items in the hashtable, without ever blocking each other. As far as we know, no wait- or lock-free algorithm for hashtables with open addressing has ever been proposed. We recently encountered a lock-free resizable hashtable with chaining based on a recursively split ordered list structure [15] which addresses the same problem but which is technically completely different.

Strictly speaking, our algorithm is lock-free. However, we call our algorithm almost wait-free since wait-freedom is only violated when a hashtable is resized, which is a relatively rare operation. Compared to sequential hashtables, our solution has the same time-complexity. It can easily be implemented using C-like languages and requires on average only constant time for insertion, deletion or accessing of elements. Regarding space requirements, our algorithm only requires one additional bit per entry in the hashtable.

As we hope we made it clear that a true problem of wait-free algorithms is that they are hard to design correctly, which even holds for apparently straightforward algorithms. Whereas human imagination generally suffices to deal with all possibilities of sequential processes or synchronized parallel processes, this appears impossible (at least to us) for lock-free algorithms. The only technique that we see fit for any but the simplest lock-free algorithms is to prove the correctness of the algorithm very precisely, and to double check this using a proof checker or theorem prover.

As a correctness notion, we take that the operations behave the same as for 'ordinary' hashtables, under any arbitrary serialization of these operations. In view of the complexity of the algorithm and its correctness properties, we turned to the theorem prover PVS [16] for mechanical support, and constructed a mechanical proof of the safety and atomicity properties of the algorithm that uses around 200 invariants (see [3]). In this extended abstract, we cannot describe the proof in any detail, but only sketch the proof obligations.

2. Specification

A hashtable with open addressing is an implementation of (partial) function, say $\mathtt{X}$, between two domains *Address* and *Value*. Thus, value v is stored at an address a in the hashtable can be described by the equality $\mathtt{X}(a) = v$. The *Address* 0 indicates the absence of the address, while *Value* **null** is the default value. In particular, $\mathtt{X}(0)$ equals **null**. We assume that from every value the address can be derived by function $\mathit{ADR} : \mathit{Value} \to \mathit{Address}$ with $\mathit{ADR}(\mathbf{null}) = 0$. Note that the existence of *ADR* is not a real restriction since one can choose to store the pair (a, v) instead of v.

One main aspect for the hashtable with open addressing is that the address in the hastable must be unique, we therefore have the following two properties:

$$\begin{array}{lcl} v = \mathbf{null} & \equiv & \mathit{ADR}(v) = 0\ , \\ \mathtt{X}(a) \neq \mathbf{null} & \Rightarrow & \mathit{ADR}(\mathtt{X}(a)) = a\ . \end{array}$$

There are four principle operations for which we give specifying descriptions below: *find*, *delete*, *insert* and *assign*. The first operation is to *find* the value currently associated with a given address. This operation yields **null** if the address has no associated value. The second operation is to *delete* the value currently associated with a given address. It fails if the address was empty, i.e. $\mathtt{X}(a) = \mathbf{null}$. The third operation is to *insert* a new value for a given address, provided the address was empty. The fourth operation is to *assign* a new value for a given address, it does the same as *insert* if the address was empty. For reasons of space we do not treat *assign* here (see [3]).

proc $\mathit{find}_S(a : \mathit{Address} \setminus \{0\}) : \mathit{Value} =$
  local $rS : \mathit{Value}$;
(fS) $\langle\ rS := \mathtt{X}(a)\ \rangle$;
return rS.

proc $\mathit{delete}_S(a : \mathit{Address} \setminus \{0\}) : \mathit{Bool} =$
  local $\mathit{sucS} : \mathit{Bool}$;
(dS) $\langle\ \mathit{sucS} := (\mathtt{X}(a) \neq \mathbf{null})$;
  if sucS **then** $\mathtt{X}(a) := \mathbf{null}$ **end** $\rangle$;
return sucS.

proc $\mathit{insert}_S(v : \mathit{Value} \setminus \{\mathbf{null}\}) : \mathit{Bool} =$
  local $\mathit{sucS} : \mathit{Bool}$; $a : \mathit{Address} := \mathit{ADR}(v)$;
(iS) $\langle\ \mathit{sucS} := (\mathtt{X}(a) = \mathbf{null})$;
  if sucS **then** $\mathtt{X}(a) := v$ **end** $\rangle$;
return sucS.

The basic correctness conditions for concurrent systems are functional correctness and atomicity, say in the sense of [14]. Functional correctness is expressed by prescribing how the procedures *find*, *delete* and *insert* affect the value of the abstract mapping $\mathtt{X}$. We use angular brackets $\langle \ldots \rangle$ to denote atomicity. The code thus specifies that the modification of $\mathtt{X}$ is executed atomically at some time between the invocation of the routine and its response. Each of these procedures has the precondition that the calling process has access to the hashtable.

We prove partial correctness of the implementation by extending it with the auxiliary variables and commands used in the specification. We regard $\mathtt{X}$ as a shared auxiliary variable and rS and *sucS* as private auxiliary variables; we augment the implementations of *find*, *delete* and *insert* with the atomic commands (fS), (dS), and (iS), respectively. We prove that the implementation of the procedure below executes its atomic specification command always precisely once and that the resulting value r or *suc* of the implementation equals the resulting value rS or *sucS* in the specification above. It follows that, by removing the implementation variables from the combined program, we obtain the specification. This removal may eliminate many atomic steps of the implementation. This is known as removal of stutterings in TLA [13] or abstraction from τ steps in process algebras. By removing the auxiliary variables, we obtain the implementation given below.

3. Algorithm

The aim is to construct a hashtable that can be accessed simultaneously by different processes in such a way that no process can passively block another process' access to the hashtable. We assume that there is a bounded number, say P, of processes that may need to concurrently interact with the hashtable. Each process is characterized by the sequence of main operations

$$(\mathit{getAccess};\ (\mathit{find} + \mathit{delete} + \mathit{insert})^*;\ \mathit{releaseAccess})^\omega$$

A process that needs to access the hashtable, first calls the procedure *getAccess* to get the current hashtable pointer. It may then invoke the procedures *find*, *delete*, and *insert* repeatedly, in an arbitrary, serial manner. A process that has access to the hashtable but is not accessing it for a while, can call *releaseAccess* to allow memory to be released.

As is well known [12], hashing with open addressing needs a special value **del** $\in$ *Value* to replace deleted values. When the current hashtable becomes full, processes will reach consensus on the choice for a new hashtable of new size to replace for, and the values except **null** and **del** will be migrated to the new hashtable. The main operations will be proceeded in the new hashtable once the migration is done. The remaining operations are all related to the migration. Since the migrating process may stop functioning during the migration, a value being migrated must be tagged by function *old* in such a way that it can be recognized by function *oldp*. We define *EValue* as an extended domain of values:

$$\mathit{EValue} = \mathit{Value} \cup \{\mathit{old}(v) \mid v \in \mathit{Value}\}$$

and introduce function *val* : *EValue* $\rightarrow$ *Value* to recall the value without the *old* tag and paticularly specify $\mathit{val}(\mathbf{del}) = \mathbf{null}$ and $\mathit{val}(\mathit{old}(\mathbf{del})) = \mathbf{null}$. The *old* tag can easily be implemented by designating one special bit in the representation of *Value*. In the sequel we write **done** for $\mathit{old}(\mathbf{null})$ or $\mathit{old}(\mathbf{del})$. Moreover, we extend the function *ADR* to domain *EValue* by $\mathit{ADR}(v) = \mathit{ADR}(\mathit{val}(v))$.

We implement function `X` via hashing with open addressing, cf. [12, 19], rather than direct chaining [15] where colliding entries are stored in a secondary list. In principle, hashing is a way to store address-value pairs in a hashtable with a length much smaller than the number of potential addresses. For the sake of the algorithm, we combine the ordinary hash function and the secondary hash function in one abstract function *key* given by:

$$\mathit{key}(a : \mathit{Address},\ l : \mathit{Nat},\ n : \mathit{Nat}) : \mathit{Nat}\ ,$$

where l is the length of the hashtable and n serves to obtain alternative locations in case of collisions. The function *key* satisfies for any address a and any number l:

$$0 \le \mathit{key}(a, l, n) < l\ ,$$
$$0 \le k < m < l \quad \Rightarrow \quad \mathit{key}(a, l, k) \ne \mathit{key}(a, l, m)\ .$$

3.1. Data Structure

Fig. 1 shows the data structure that is used by the processes. A hashtable is of type *Hashtable*, where the field `size` indicates the size of the hashtable, the field `bound` the maximal number of places that can be occupied before refreshing the hashtable, `bound` must be less than `size`. Both are constants that are set when the hashtable is created and tuned for optimal performance. The field `occ` gives the number of occupied positions in the hashtable, while the field `dels` gives the number of deleted positions with the purpose of allowing our hashtable to shrink. If h is a pointer to a hashtable, we write h.`size`, h.`occ`, h.`dels` and h.`bound` to access these fields of the hashtable. We write h.`table`$[i]$ to access the i^{th} *EValue* in the hashtable. Apart from the *current* hashtable (pointed to by `H(currInd)`), which is the main representative of the specification variable `X`, we have to deal with *old* hashtables, which were in use before the current one, and *new* hashtables, which can be created after the current one. There can be $2P$ hashtables around, because each process can simultaneously be accessing one hashtable and attempting to create a new one.

The basic idea is to count the number of processes that are using a hashtable, by means of a counter `busy`. The hashtable can be thrown away when `busy` is set to 0. An important observation is that `busy` cannot be stored as part of the hashtable, in the same way as the variables `size`, `occ` and `bound` above. The reason for this is that a process can attempt to access the current hashtable by increasing its `busy` counter. However, just before it wants to write the new value for `busy` it falls asleep. When the process wakes up the hashtable might have been deleted and the process would be writing at a random place in memory.

constant P = number of processes
type *Hashtable*: **record=**
  `size`, `bound`, `occ`, `dels`: *Nat*;
  `table` : **array**$[0..$`size`$-1]$ **of** *EValue* ;
 end
shared variables:
 `currInd` : $[1..2P]$;
 `H` : **array**$[1..2P]$ **of pointer to** *Hashtable*;
 `prot,busy` : **array**$[1..2P]$ **of** *Nat*;
 `next` : **array**$[1..P]$ **of** $[0..2P]$;
public initialization:
 all shared variables are initially 0 or **nil** except:
  `currInd` can be any value in $[1..2P]$;
  pointer `H[currInd]` points to the current hashtable,
  which is valid and empty with `size` $>$ `bound` $+ 2P$;
  there is no other valid hashtable;
  `prot[currInd]` $= 1$; `busy[currInd]` $= 1$.
private variables:
 index : $[1..2P]$; *pc* : *Nat*;
private initialization:
 index can be any value in $[1..2P]$; $pc = 0$.

Figure 1. Data Structure.

This forces us to use separate arrays `H` and `busy` to store the pointers to hashtables and the `busy` counters. As indicated, we also need arrays `prot` and `next`. The variable `next`$[i]$ points to the next hashtable to which the contents of hashtable `H`$[i]$ is being copied. If `next`$[i]$ equals 0, this means that there is no next hashtable. The variable `prot`$[i]$ is used to guard the variables `busy`$[i]$, `next`$[i]$ and `H`$[i]$ against being reused for a new hashtable, before all processes have discarded them.

3.2. Primary Procedures

We provide the code for the primary procedures, which match directly with the procedures in the specification. Every process has a private variable *index* containing what it regards as the current hashtable. Private variable *pc* stands for program counter. Each process, numbered from 1 up to P, is a sequential program comprised of statements that are thought to be executed atomically. Except for some special commands (offered by machine architectures) such as compare&swap and test&set enclosed by angular brackets $\langle \ldots \rangle$, all actions on variables are separated into different atomic accesses, and we labeled each group of statements in which at most one access to a public variable takes place with a number. The labels are those chosen in the PVS code, and are therefore not completely consecutive.

In order to prove correctness, we added instructions that modify auxiliary variables (between braces $\{\ldots\}$). Since auxiliary variables are only used to facilitate the proof of correctness, they can be assumed to be touched instantaneously without violation of the atomicity restriction, and can therefore harmlessly be added to an atomic statement within the angular brackets. As procedure calls only modify private control data, procedure headers are not labeled themselves, but their bodies usually have numbered atomic statements.

The main operations are given in Fig. 2 and 3. Finding an address in a hashtable with open addressing requires a linear search over the possible hash keys until the address or an empty slot is found. The main complication is that the process has to join the migration activity if it encounters an entry **done**, which indicates that another process started migrating. So, the new hashtable must be located, which is carried out by the procedure *refresh* (see section 3.3). Note that it is not necessary to assist in migrating when a value $old(v)$ is encountered. In this case it is safe to let *find* continue.

With the help of PVS, we have proved that when some process is executing *find*, no other process can *delete* or *insert* an entry associated with the same address in the region where the process has already searched, and thus violate the views of data for the process. We have also proved that, when the `bound` of the new hashtable is tuned properly before use (see section 3.3), there exist at least one **null** or **done** entry in any valid hashtable. Hence, the local variable n in the procedure *find* will never go beyond the size of the hashtable.

Deletion is similar to finding. Since r is a local variable to the procedure *delete* and line 18b is a so-called compare&swap instruction, we can take 18a and 18b as two parts of atomic instruction 18. If the entry is outdated, the process joins the others to complete the migration and then proceeds with the deletion in the new table. Since we postpone the increment of h.`dels` until line 25, the field `dels` actually reflects a lower bound of the number of positions deleted in the hashtable. Line 25 is a so-called fetch&add instruction.

Basically, the procedure for insertion is the standard algorithm for insertion. Any deleted value can not be reused in a subsequent insertion. Notable is line 28 where the current process finds the current hashtable too full, and orders a new table to be made. When the new hashtable has been located at line 37, we do not need to re-evaluate the field `occ` since we have proved that field `size` is at least $2P$ more than field `occ`. Instruction 35b is a test&set instruction, a simpler version of compare&swap.

We did not force the above procedures to check if the hashtable pointer is valid, but it should be guaranteed that no process can inspect invalid contents. Since `prot`[`currInd`] and `busy`[`currInd`] are always positive in our system (see [3]), procedure *getAccess* serves to initialize the private variable *index* in such a way that the hashtable pointer `H`[*index*] is valid and that it is not in-

```
    proc find(a : Address \ {0}) : Value =
       local r : EValue ; n, l : Nat ; h : pointer to Hashtable ;
5:     h := H[index] ; n := 0 ;
6:     l := h.size ;
       repeat
7:        ⟨ r := h.table[key(a, l, n)] ;
           { if r = null ∨ a = ADR(r) then (fS) fi } ⟩
8:        if r = done then refresh() ;
10:          h := H[index] ; n := 0 ;
11:          l := h.size ;
          else n++ ; fi
13:    until r = null ∨ a = ADR(r) ;
14: return val(r) .

    proc delete(a : Address \ {0}) : Bool =
       local r : EValue ; k, l, n : Nat ; h : pointer to Hashtable ;
             suc : Bool ;
15:    h := H[index] ; suc := false ;
16:    l := h.size ; n := 0 ;
       repeat
17:       k := key(a, l, n) ;
          ⟨ r := h.table[k] ; { if r = null then (dS) fi } ⟩
18a:      if oldp(r) then refresh() ;
20:          h := H[index] ;
21:          l := h.size ; n := 0 ;
          elseif a = ADR(r) then
18b:         ⟨ if r = h.table[k] then h.table[k] := del ;
               suc := true ; { (dS) } fi ⟩
          else n++ ; fi
       until suc ∨ r = null ;
25:    if suc then ⟨ h.dels++ ; ⟩ fi
26: return suc .

    proc insert(v : Value \ {null}) : Bool =
       local r : EValue ; k, l, n : Nat ; h : pointer to Hashtable ;
             suc : Bool ; a : Address ;
27:    a := ADR(v) ; h := H[index] ;
28:    if h.occ > h.bound then newTable() ;
30:       h := H[index] ; fi
31:    n := 0 ; l := h.size ; suc := false ;
       repeat
32:       k := key(a, l, n) ;
33:       ⟨ r := h.table[k] ;
           { if a = ADR(r) then (iS) fi } ⟩
35a:      if oldp(r) then refresh() ;
36:          h := H[index] ;
37:          n := 0 ; l := h.size ;
          elseif r = null then
35b:         ⟨ if h.table[k] = null then suc := true ;
                  h.table[k] := v ; { (iS) } fi ⟩
          else n++ ; fi
       until suc ∨ a = ADR(r) ;
41:    if suc then ⟨ h.occ++ ; ⟩ fi
42: return suc .
```

Figure 2. Procedure *find*, *delete*, *insert*.

```
    proc getAccess() =
       loop
59:       index := currInd;
60:       ⟨ prot[index]++ ; ⟩
61:       if index = currInd then
62:          ⟨ busy[index]++ ; ⟩
63:          if index = currInd then return
             else releaseAccess(index) ; fi
65:       else ⟨ prot[index]-- ; ⟩ fi
    end end .

    proc releaseAccess(i : 1 .. 2P) =
       local h : pointer to Hashtable ;
67:    h := H[i] ;
68:    ⟨ busy[i]-- ; ⟩
69:    if h ≠ nil ∧ busy[i] = 0 then
70:       ⟨ if H[i] = h then H[i] := nil ; ⟩
71:          deAllocate(h) ; fi fi
72:    ⟨ prot[i]-- ; ⟩
    end .
```

Figure 3. Procedure *getAccess*, *releaseAccess*.

advertently destroyed (both requirements are protected by busy[*index*]) or used to create a new hashtable (this requirement is protected by prot[*index*]) before the process actively releases it. Both prot[*index*] and busy[*index*] are used primarily as counters with atomic increments and decrements since they are both shared.

In procedure *releaseAccess*, the process releases its claim on the hastable pointed to by H[i] by decrementing busy[i] and prot[i]. When busy[i] becomes 0, the hashtable itself must be deallocated. A bigger atomic command (i.e. compare& swap) in line 70 is needed to preclude that the hashtable is deallocated more than once. Indeed, in line 71, *deAllocate* is called only for allocated memory. It is incorrect to replace the compare&swap statement 70 by the assignment H[i] := nil. This replacement would introduce a race condition and hashtable H[i] could be deallocated twice.

3.3. Procedures for Migration

The remaining operations are shown in Fig. 4 and Fig. 5. Procedure *newTable* is called when the number of occupied positions in the hashtable exceeds the bound. It first searches for a free index i, say by round robin. We represent this by a nondeterministic choice. Then it tries to allocate a new empty hashtable in line 82. If several processes call *newTable* concurrently, they need to reach consensus (in line 84) on the choice of the next hashtable. A hashtable newly allocated by a late process will not be used and must be deallocated again. In 78, we use an atomic

```
     proc newTable() =
        local i : 1 .. 2P ;
77:     while next[index] = 0 do
78:        choose i ∈ 1 .. 2P ;
           ⟨ if prot[i] = 0 then prot[i] := 1 ; ⟩
81:           busy[i] := 1 ;
82:           choose size, bound ;
              H[i] := allocate(size, bound) ;
83:           next[i] := 0 ;
84:           ⟨ if next[index] = 0 then next[index] := i ;
               else releaseAccess(i) ; fi ⟩ fi
        end ;
        refresh() ;
     end .

     proc refresh() =
90:     if index ≠ currInd then
           releaseAccess(index) ; getAccess()
        else migrate() ; fi
     end .

     proc migrate() =
        local i : 0 .. 2P ; h : pointer to Hashtable ;
94:     i := next[index] ;
95:     ⟨ prot[i]++ ; ⟩
97:     if index ≠ currInd then
98:        ⟨ prot[i]-- ; ⟩
        else
99:        ⟨ busy[i]++ ; ⟩
100:       h := H[i] ;
101:       if index = currInd then
              moveContents(H[index], h) ;
103:          ⟨ if index = currInd then currInd := i ; ⟩
104:             ⟨ busy[index]-- ; ⟩
105:             ⟨ prot[index]-- ; ⟩ fi fi
           releaseAccess(i) ; fi
     end .

     proc moveContents(from, to : pointer to Hashtable) =
        local i : Nat ; v : EValue ; toBeMoved : set of Nat ;
109:    toBeMoved := {0, . . . , from.size − 1} ;
110:    while currInd = index ∧ toBeMoved ≠ ∅ do
111:       choose i ∈ toBeMoved ; v := from.table[i] ;
112:       if v = done then
              toBeMoved := toBeMoved − {i} ;
           else
114:          ⟨ if v = from.table[i] then
                 from.table[i] := old(val(v)) ; ⟩
116:             if val(v) ≠ null then
                    moveElement(val(v), to) ; fi
117:             from.table[i] := done ;
118:             toBeMoved := toBeMoved − {i} ; fi fi
     end end .
```

Figure 4. Procedure *newTable*, *refresh*, *migrate* **and** *moveContents*.

test&set instruction. Indeed, separation of this instruction in two atomic instructions is incorrect, since that would allow two processes to grab the same index i concurrently.

Field `occ` and field `dels` in the new hashtable are initially 0 since the new hashtable is empty. Due to some particular scenario (see [3]) and our proof obligations, in command 82, the `bound` is required to be chosen to be greater than $\mathtt{H}[\mathit{index}].\mathtt{bound} - \mathtt{H}[\mathit{index}].\mathtt{dels} + 2P$ and the `size` to be more than $\mathtt{bound} + 2P$. These conditions are proved to be sufficient for the safety conditions that will be described later. It may be useful to make field `size` even larger than $\mathtt{bound} + 2P$ to avoid too many collisions, e.g. with a constraint $\mathtt{size} \geq \alpha \cdot \mathtt{bound}$ for some $\alpha > 1$.

In order to avoid that a process starts migration of an old hashtable, we encapsulate it in a procedure *refresh*. When *index* is outdated, it is necessary that the process calls *releaseAccess* to abandon its hashtable and *getAccess* to acquire the pointer to the current hashtable. Otherwise, the process can join the migration.

After the choice of the new hashtable, procedure *migrate* serves to transfer the contents in the current hashtable to the new hashtable by means of *moveContents* and to update the current hashtable pointer afterwards. Migration is complete when at least one of the (parallel) calls to *migrate* has terminated. Line 103 contains a compare&swap instruction to update the current hashtable pointer when some process finds that the migration is finished while `currInd` is still identical to its *index*. This means that i is still used for the index of the next current hashtable. The increments of $\mathtt{prot}[i]$ and $\mathtt{busy}[i]$ here are needed to protect the next hashtable. The decrements serve to avoid memory loss.

Procedure *moveContents* has the task to move the contents from the current hashtable to the next current hashtable. All processes that have access to a hashtable, can participate in this migration, until they can use the next current hashtable. We have to take care that delayed actions on the current hashtable and the new hashtable are carried out or abandoned correctly. Note that the value is tagged as *old* before it is migrated. After tagging, the value cannot be deleted or assigned until the migration has been completed. Tagging must be done atomically, since otherwise an interleaving deletion may be lost. The value is made **done** when it has been copied to the new hashtable. In this way other processes need not wait for this process to complete procedure *moveElement* of Fig. 5, but can help with the migration of the value if needed.

The processes involved in the same migration should not use the same strategy for choosing i in line 111, since it is advantageous that *moveElement* is called often with different values. They may exchange information: any of them may replace its set *toBeMoved* by the intersection of that set with the set *toBeMoved* of another one. We do not give a preferred strategy here, one can refer to algorithms for the

proc $moveElement(v : Value \setminus \{\mathbf{null}\},$
$\quad to : \mathbf{pointer\ to}\ Hashtable) =$
local $a : Address\ ;\ k, m, n : Nat\ ;\ w : EValue\ ; b : Bool\ ;$
120: $a := ADR(v)\ ;\ m := to.\mathtt{size}\ ;\ n := 0\ ;$
repeat
121: $k := key(a, m, n)\ ;\ w := to.\mathtt{table}[k]\ ;$
if $w =$ **null then**
123: $\langle$ **if** $to.\mathtt{table}[k] =$ **null then**
$to.\mathtt{table}[k] := v\ ;\ b := true\ ;$ **fi** $\rangle$
else $n{+}{+}$; **fi** ;
125: **until** $b \vee a = ADR(w) \vee \mathtt{currInd} \neq index$;
126: **if** b **then** $\langle\ to.\mathtt{occ}{+}{+}\ ;\ \rangle$ **fi**
end .

Figure 5. Procedure *moveElement***.**

write-all problem [4, 11].

To illustrate the subtlety of the algorithm, we note that one could propose to replace the tests $oldp(r)$ in the lines 18a and 35a by $\mathtt{next}[index] \neq 0$, followed by an elimination of function *old*. In this way, one might hope to replace the set *EValue* by $Value \cup \{\mathbf{done}\}$. This is incorrect, however, since one then needs a bigger atomic statement, or procedure *moveContents* can disastrously interfere with procedure *delete* (and also with *assign*, see [3]).

Procedure *moveElement* copies the non-null value v to the new hashtable. Migration requires that every value except **null** and **del** in the current hashtable is migrated only once to a unique position in the new hashtable, for, otherwise, the main property of open addressing would be violated. It has been proved that no process is able to insert a value in the new hashtable when its *index* differs from `currInd`. We have also proved that there exists at least one **null** entry in the new hashtable while the current hashtable pointer is not updated yet, Hence the local variable n in the procedure *moveElement* never goes beyond the size of the hashtable, and the termination is thus guaranteed.

4. Correctness

In order to ensure that our algorithm is correct and reliable, we have to justify safety and progress. The safety property consisting of around 200 invariants has been verified with PVS. The main aspect of safety is functional correctness. Functional correctness of *find*, *delete* and *insert* is the condition that the result of the implementation is the same as the result of the specification (fS), (dS) and (iS). This is expressed by the required invariants:

Co1: $pc = 14 \Rightarrow val(r_{fi}) = rS_{fi}$
Co2: $pc \in \{25, 26\} \Rightarrow suc_{del} = sucS_{del}$
Co3: $pc \in \{41, 42\} \Rightarrow suc_{ins} = sucS_{ins}$

where the subscript indicates the name of the procedure each local variable belongs to.

According to the definition of atomicity in [14], atomicity means that in each of the procedures *find*, *delete* and *insert* the specifying command (fS), (dS) and (iS) is executed precisely once, respectively. This has been proved formally in [3]. We interpret absence of memory loss to mean that the number of allocated hashtables at any given time is bounded (by $2P$). In [3], we have also proved the absence of memory loss. Moreover, if the size of the hashtable is bounded, all counters are bounded.

We now turn to the discussion of progress. As announced, the algorithm is lock-free and almost wait-free. The first point to note is that, according to the invariants described in [3], the primary procedures *find*, *delete* and *insert* are loops bounded by the `size` of the hashtable, so they are wait-free unless they are reset infinitely often. This reset only occurs immediately after the call of *refresh* (in lines 8, 18a, 35a).

Procedure *getAccess* is not wait-free. When the active clients keep changing the current index faster than the new client can observe it, the accessing client is doomed to starvation. It may be possible to make *getAccess* wait-free by introducing a queue for the accessing clients which is emptied by a process in *newTable*. The accessing clients must however also be able to enter autonomously. This would at least add another layer of complications. We therefore prefer to treat this failure of wait-freedom as a performance issue that can be dealt with in practice by tuning the sizes of the hashtables.

It is clear that *releaseAccess* is wait-free. It follows that the wait-freedom of *migrate* depends on wait-freedom of *moveContents*. Wait-freedom of *moveContents* depends on wait-freedom of *moveElement*. As the loop of *moveElement* is also bounded [3], this concludes the sketch that *migrate* is wait-free. The main part of procedure *newTable* is wait-free, see [3].

Therefore, if we assume that the current hashtable is not updated too often, the primary procedures are wait-free. Under these circumstances, *getAccess* is also wait-free, and then everything is wait-free.

5. Conclusions

Wait-free shared data objects are implemented without any unbounded busy-waiting loops or idle-waiting primitives. They are inherently resilient to halting failures and permit maximum parallelism. We have presented a new practical algorithm, which is almost wait-free, for concurrently accessible hashtables, which promises more robust performance and reliability than a conventional lock-based implementation. Moreover, the new algorithm is dynamic in the sense that it allows the hashtable to grow and shrink as needed.

The algorithm scales up linearly with the number of processes, provided the function *key* and the selection of i in line 111 are defined well. This is confirmed by some experiments where random values were stored, retrieved and deleted from the hashtable. These experiments indicated that 10^6 insertions, deletions and finds per second and per processor are possible on an SGI powerchallenge with 250Mhz R12000 processors. This figure should be taken as a rough indicator, as the performance of parallel processing is very much influenced by the machine architecture, the relative sizes of data structures compared to sizes of caches, and even the scheduling of processes on processors.

The correctness proof for our algorithm is noteworthy because of the extreme effort it took to finish it. Formal deduction by human-guided theorem proving can, in principle, verify any correct design, but doing so may require unreasonable amounts of effort, time, or skill. Though PVS provided great help for managing and reusing the proofs, we have to admit that the verification for our algorithm was very complicated due to the complexity of our algorithm. The total verification effort can roughly be estimated to consist of two man years excluding the effort in determining the algorithm and writing the documentation. 'Fortunately', we received compensation as approximately a dozen errors in the algorithm were found in this way, that would be extremely hard to find in any other manner.

The whole proof contains around 200 invariants. Without suitable tool support like PVS, we even doubt if it would be possible to complete the proof of such size and complexity. The complete version of the PVS specifications and the whole proof scripts can be found at [10].

References

[1] Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. J. ACM 37, 1990, pp. 524–548.

[2] Bar-Noy, A., Dolev, D.: Shared-memory vs. message–passing in an asynchronous distributed environment. In Proc. 8th ACM Symp. on principles of distributed computing, 1989, pp. 307–318.

[3] H. Gao, J.F. Groote and W.H. Hesselink.: Efficient almost wait-free parallel accessible dynamic hashtables. Technical Report CS-Report 03-03, Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 2003. http://www.win.tue.nl/∼jfg/articles/CS-Report03-03.pdf.

[4] Groote, J.F., Hesselink, W.H., Mauw, S., Vermeulen, R.: An algorithm for the asynchronous write–all problem based on process collision. Distributed Computing 14, 2001, pp. 75–81.

[5] Herlihy, M.P.: Wait–free synchronization. ACM Trans. on Program. Languages and Systems 13, 1991, pp. 124–149.

[6] Herlihy, M.P. and Moss, J.E.B.: Lock–free garbage collection for multiprocessors. IEEE Transactions on Parallel and Distributed Systems 3, 1992, pp. 304–311.

[7] Hesselink, W.H.: Wait–free linearization with a mechanical proof. Distributed Computing 9, 1995, pp. 21–36.

[8] Hesselink, W.H.: Bounded Delay for a Free Address. Acta Informatica 33, 1996, pp. 233–254.

[9] Hesselink, W.H., Groote, J.F.: Wait-free concurrent memory management by Create, and Read until Deletion (CaRuD). Distributed Computing 14, 2001, pp. 31–39.

[10] http://www.cs.rug.nl/∼wim/mechver/hashtable.

[11] Kanellakis, P.C., Shvartsman, A.A.: Fault-tolerant parallel computation. Kluwer Academic Publishers, 1997.

[12] Knuth, D.E.: The Art of Computer Programming. Part 3, Sorting and searching. Addison-Wesley, 1973.

[13] Lamport, L.: The temporal logic of actions. ACM Trans. on Programming Languages and Systems 16, 1994, pp. 872–923.

[14] Lynch, N.A.: Distributed Algorithms. Morgan Kaufman, San Francisco, 1996.

[15] Ori Shalev, Nir Shavit: Split–Odered Lists - Lock–free Resizable Hash Tables. Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing (PODC 2003) Boston, Massachusetts, 2003.

[16] Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Version 2.4 (2001). System Guide, Prover Guide, PVS Language Reference. http://pvs.csl.sri.com.

[17] Valois, J.D.: Lock-free linked lists using compare-and-swap. Proceedings of the 14th Annual Principles of Distributed Computing, 1995, pp. 214–222.

[18] Valois, J.D.: Implementing Lock–Free Queues, Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, October 1994, pp. 64–69.

[19] Wirth, N.: Algorithms + Data Structures = Programs. Prentice Hall, 1976.