Int J Parallel Prog (2017) 45:421–448
DOI 10.1007/s10766-015-0376-3

A Wait-Free Hash Map

Pierre Laborde[1] · **Steven Feldman**[1] · **Damian Dechev**[1]

Received: 13 February 2014 / Accepted: 5 August 2015 / Published online: 19 August 2015

Abstract In this work we present the first design and implementation of a wait-free hash map. Our multiprocessor data structure allows a large number of threads to concurrently insert, get, and remove information. Wait-freedom means that all threads make progress in a finite amount of time—an attribute that can be critical in real-time environments. This is opposed to the traditional blocking implementations of shared data structures which suffer from the negative impact of deadlock and related correctness and performance issues. We only use atomic operations that are provided by the hardware; therefore, our hash map can be utilized by a variety of data-intensive applications including those within the domains of embedded systems and supercomputers. The challenges of providing this guarantee make the design and implementation of wait-free objects difficult. As such, there are few wait-free data structures described in the literature; in particular, there are no wait-free hash maps. It often becomes necessary to sacrifice performance in order to achieve wait-freedom. However, our experimental evaluation shows that our hash map design is, on average, 7 times faster than a traditional blocking design. Our solution outperforms the best available alternative non-blocking designs in a large majority of cases, typically by a factor of 15 or higher.

✉ Pierre Laborde
pierrelaborde@knights.ucf.edu

Steven Feldman
Feldman@knights.ucf.edu

Damian Dechev
dechev@eecs.ucf.edu

[1] University of Central Florida, Orlando, FL, USA

Keywords Lock-free · Wait-free · Non-blocking · Hash map · Data Structures · Parallel programming · Concurrency

1 Introduction

Our design is motivated by the need for applications and algorithms to change and adapt as modern architectures evolve. These adaptations have become increasingly difficult for developers as they are required to effectively manage an ever-growing variety of resources such as a high degree of parallelism, single-chip multi-processors, and the deep hierarchies of shared and distributed memories. Developers writing concurrent code face challenges not known in sequential programming, most importantly, the correct manipulation of shared data. The new C++ standard, C++11, includes a large number of concurrency features, such as atomic operations. However, C++11 still does not offer a standard collection of parallel multiprocessor data structures. The standard collection of data structures and algorithms in C++11 is the inherently sequential Standard Template Library (STL).

Currently, the most common synchronization technique is the use of mutual exclusion locks. Blocking synchronization can seriously affect the performance of an application by diminishing its parallelism [13]. The behavior of mutual exclusion locks can sometimes be optimized by using a fine-grained locking scheme [15,27] or context-switching. However, the interdependence of processes implied by the use of locks, even efficient locks, introduces the dangers of deadlock, livelock, starvation, and priority inversion—our design avoids these drawbacks.

This paper is extended from a conference version [8].

1.1 Our Approach

The main goal of our design is to deliver a hash map that provides both *safety* and *high performance* for multi-processor applications.

The hardest problem encountered while developing a parallel hash map is how to perform a global resize, the process of redistributing the elements in a hash map that occurs when adding new buckets. The negative impact of blocking synchronization is multiplied during a global resize, because all threads will be forced to wait on the thread that is performing the involved process of resizing the hash map and redistributing the elements. Our wait-free implementation avoids global resizes through new array allocation. By allowing concurrent expansion this structure is free from the overhead of an explicit resize, which facilitates concurrent operations.

The presented design includes dynamic hashing, the use of sub-arrays within the hash map data structure [20]; which, in combination with perfect hashing, means that each element has a unique final, as well as current, position. It is important to note that the perfect hash function required by our hash map is trivial to realize as any hash function that permutes the bits of the key is suitable. This is possible because of our approach to the hash function; we require that it produces hash values that are equal in size to that of the key. We know that if we expand the hash map a fixed number of times there can be no collision as duplicate keys are not provided for in the

standard semantics of a hash map. The aforementioned properties are used to achieve the following design goals:

(a) *Wait-free*: a progress guarantee, provided by our data structure, that requires all threads to complete their operations in a finite number of steps [13].
(b) *Linearizable*: a correctness property that requires seemingly instantaneous execution of every method call; the point in time that this appears to occur is called a linearization point, which implies that the real-time ordering of calls are retained [13].
(c) *High performance*: our wait-free hash map design outperforms, by a factor of 15 or more, state of the art non-blocking designs. Our design performs a factor of 7 or greater faster than a standard blocking approach.
(d) *Safety*: our design goals help us achieve a high degree of safety; our design avoids the hazards of lock-based designs.

The rest of this work is organized as follows: Sect. 2 briefly introduces the fundamental concepts of non-blocking synchronization, Sect. 3 discusses related work, Sect. 4 presents the algorithms of our wait-free hash map design, Sect. 5 presents an informal proof of correctness, Sect. 6 offers a discussion of our performance evaluation, Sect. 7 provides an overview of the practical impact of our work, and Sect. 8 offers conclusions and a discussion of our future work.

2 Background

A hash map is a data container that uses a hash function to map a set of identifying values, known as keys, to their associated values [3]. The standard interface of a hash map consists of three main operations: put, get, and remove; each operation has an average time complexity of $O(1)$.

Standard hash maps used in software development are designed to work in sequential environments, where only one process can modify the data at any moment in time. In a concurrent environment, there is no guarantee that the hash map will be in a consistent state when more than one process attempts to modify it; one potential problem is that a newer value may be replaced by an older value. The solution to these issues was the development of lock-based hash maps [1].

Each process that wished to modify the hash map would have to lock the entire hash map. If the hash map was already locked, then all other processes needed to wait until the holder of the lock released it. This led to performance bottlenecks as more parallel processes were added to the system, because these processes would have to wait on the others [28]. Eventually, fine-grained locking schemes were also proposed, but even these approaches suffered from the negative consequences of blocking synchronization [15].

As defined by Herlihy et al. [13,14], a concurrent object is *lock-free* if it guarantees that *some* process in the system makes progress in a *finite* number of steps. An object that guarantees that *each* process makes progress in a *finite* number of steps is defined as *wait-free* [13]. By applying atomic primitives such as CAS, non-blocking algorithms, including those that are lock-free and wait-free, implement a number of

techniques such as optimistic speculation and thread collaboration to provide for their strict progress guarantees. As a result of these requirements, the practical implementation of non-blocking containers is known to be difficult.

A common problem with non-blocking designs is called the ABA problem. This refers to the following situation:

(1) One thread reads a memory address, and sees A.
(2) Then, a context switch occurs.
(3) The new thread changes the value to B, and the back to A.
(4) The original thread resumes, and does not know that any change has occurred.
(5) This could cause a semantic violation—an example of the ABA problem.

3 Related Work

Research into the design of non-blocking data structures includes: linked-lists [11,23]; queues [25,26,32]; stacks [12,26]; hash maps [10,23,26]; hash tables [30]; binary search trees [9], and vectors [5].

There are no pre-existing wait-free hash maps in the literature; as such, the related work that we discuss consists entirely of lock-free designs. In [23], Michael presents a lock-free hash map that uses linked-lists to resolve collisions; this design differs from ours in that it does not guarantee constant-time for operations after a resize is performed [23,30]. In [10], Gao et al. present an openly-addressed hash map that is *almost* wait-free; it degrades in performance to lock-free during a resize.

In [30], Shalev and Shavit present a linked-list structure that uses pointers as shortcuts to logical buckets that allow the structure to function as a hash table. In contrast to our design, the work by Shalev and Shavit does not present a hash map and it is lock-free. There was a single claim of a wait-free hash map that appeared as a presentation by Cliff Click [2]; the author now claims lock-freedom. Moreover, the work by Click was not published. A popular concurrent hash map that is part of Intel's Threading Building Blocks (TBB) [15] library is claimed to be lock-free, but is also unpublished.

4 Algorithms

In this section we define a semantic model of the hash map's operations, address concerns related to memory management, and provide a description of the design and the applied implementation techniques. The presented algorithms have been implemented, in both ISO C and ISO C++, and designed for execution on an ordinary, multi-threaded, shared-memory system; we require only that it supports atomic single-word read, write, and CAS instructions.

4.1 Structure and Definition

Our hash map is a multi-level array which has a structure similar to a tree; this is shown in Fig. 1. Our multi-level array differs from a tree in that each position on the tree could hold an array of nodes or a single node. A position that holds a single node

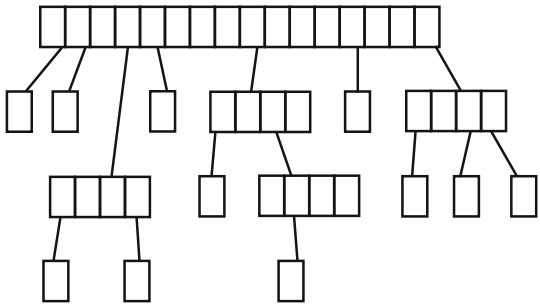


Fig. 1 An illustration of the structure of the hash map

is a `dataNode` which holds the hash value of a key and the value that is associated with that key; it is a simple struct holding two variables. Since a `dataNode` is at least two memory words we cannot read it atomically, so we must have a way to prevent interference with nodes that are being read or are otherwise in use; we call our method of doing this, "watching" (see Sect. 4.4). A `dataNode` in our multi-level array could be marked. A `markedDataNode` refers to a pointer to a `dataNode` that has been bitmarked at the least significant bit (LSB) of the pointer to the node. This signifies that this `dataNode` is contended. An expansion must occur at this node; any thread that sees this `markedDataNode` will try to replace it with an `arrayNode` ; which is a position that holds an array of nodes. The pointer to an `arrayNode` is differentiated from that of a pointer to a `dataNode` by a bitmark on the second-least significant bit.

Our multi-level array is similar to a tree in that we keep a pointer to the root, which is a memory array that we call `head`. The length of the `head` memory array is unique, whereas every other `arrayNode` has a uniform length; a normal `arrayNode` has a fixed power-of-two length equal to the binary logarithm of a variable called `arrayLength`. The maximum depth of the tree, `maxDepth`, is the maximum number of pointers that must be followed to reach any node. We define `currentDepth` as the number of memory arrays that we need to traverse to reach the `arrayNode` on which we need to operate; this is initially one, because of `head`.

Our approach to the structure of the hash map uses an extensible hashing scheme; we treat the hash value as a bit string and rehash incrementally [7]. We use `arrayLength` to determine how many bits are necessary to ascertain the location at which a `dataNode` should be placed within the `arrayNode`. The hashed key is expressed as a continuous list of `arrayPow`-bit sequences, where `arrayPow` is the binary logarithm of the `arrayLength`; e.g. $A-B-C-D$, where A is the first `arrayPow`-bit sequence, B is the next `arrayPow`-bit sequence, and so on; these represent positions on different `arrayNodes`. These bit sequences are isolated using logical shifts. We use `R` to designate the number of bits to shift right, in order to isolate the position in the `arrayNode` that is of interest. `R` is equal to $\log_2$ `arrayLength*currentDepth`. For example, in a memory array of length $64 = 2^6$, we would take `R`$= 6$ bits for each successive `arrayNode`.

The total number of arrays is bounded by the number of bits in the key (which is stored in a variable called `keySize`) divided by the number of bits needed to represent the length of each array. For example, with a 32-bit key and an `arrayLength` of 64,

we have a `maxDepth` of 6, because $\lceil 32/\log_2 64 \rceil = 6$. This places no limit on the total number of elements that can be stored in the data structure; the hash map expands to hold all unique keys that can be represented by the number of bits in the key (even beyond the machine's word size). We have tested with multiword keys, such as the 20 bytes needed for SHA1. Neither an `arrayNode` nor a `markedDataNode` can be present in an `arrayNode` whose `currentDepth` is equal to `maxDepth`, because no hash collisions can occur there.

4.2 Traversal

Traversing the hash map is done by performing a right logical shift on the hashed key to preserve `R` bits, and examining the pointer at that position on the current memory array. If the pointer stores the address of an `arrayNode`, then the `currentDepth` increases by one, and that position on the new memory array is examined.

We discuss the traversal of the hash map using Fig. 2 as an illustration of this process. In our example, the arrayNodes have a length of four, which means that exactly two bits are needed to determine where to store our `dataNode` on any particular `arrayNode`, except for `head` which has a larger size than every other `arrayNode` (see Sect. 4.1). The hashed key is expressed as a finite list of two-bit sequences e.g. $A - B - C$, where C is the first three-bit sequence, and so on; these sequences represent positions at various depths.

For example, if we need to find the key 0-4-2, in the hash map shown in Fig. 2, then we first need to hash the key. We assume that this operation yields 2-3-1. To find 2-3-1 we first take the right-most set of bits, and go to that position on `head`. We see that this is an `arrayNode`, so we take the next set of bits which leads us to examine position 3 on this `arrayNode`. This position is also an `arrayNode`, so we take the next set of bits which equal 2, and examine that position on this `arrayNode`. That position is a `dataNode`, so we compare its hashed key to the hashed key that we are searching for. The comparison reveals that the hash values are both equal to 2-3-1, so we return the value associated with this `dataNode`.

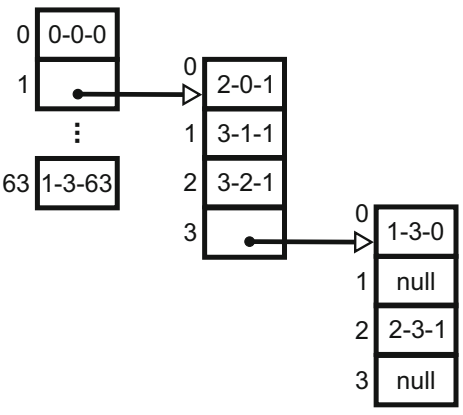


Fig. 2 An example of data stored in the hash map (values not shown)

4.3 Main Functions

In this section we provide a brief overview of the main operations implemented by our hash map. Unless otherwise noted, all line numbers refer to the current algorithm being discussed. In other sections of the paper, the main functions are referred to by the first letter of the function name followed by the line number of interest; supporting functions are referred to by their full name. In all algorithms, `local` is the name of the `arrayNode` that an operation is working on and `pos` is the position on `local` that is of interest. The variable `failCount` is a thread-local counter that is incremented whenever a CAS fails and the thread must retry its attempt to update the hash map. Instances of this variable are compared to the `maxFailCount` which is a user-defined constant used to bound the maximum number of times that a thread retries an operation after a CAS operation fails. If this bound is reached, then an expansion is forced at the position that the failing operation is attempting to modify.

The CAS operation that we use is part of C++11; the function that we use returns the value that the memory address held before the execution of the operation. If our functions are implemented in a system that does not have a sequentially consistent memory model, then memory fences are needed to preserve the relative order of critical memory accesses [23].

4.3.1 Algorithm 1: `insert (key, value)`

The `insert` function is used to insert a key-value pair into the hash map. The function returns true if the key is not in the hash map, and false if the key is already there; this allows us to prevent the user from performing unintended overwrites of elements in the hash map. We provide an `update` operation for the case wherein a user would like to change the value that is associated with a key that is already in the hash map (see Sect. 4.3.2).

An `insert` operation traverses the hash map as described in Sect. 4.2 until it finds a position that is `null` or that contains a `dataNode`. If the position is `null`, then a CAS is performed; this is shown on line 13. If the CAS is successful, then the function returns true. If a `dataNode` whose key matches the key that is being inserted, is encountered during the traversal, then the function returns false. If it is a `dataNode` whose key is different, then the thread calls `expandMap` at the position (resolving the hash collision); if the expansion is successful, then the thread continues its traversal from the new `arrayNode` that was added.

If the CAS at line 13 failed, then the CAS operation has returned either a `dataNode` or an `arrayNode`. If an `arrayNode` was returned, then the thread continues traversal from the `arrayNode`. If the result is a `dataNode` whose key matches the key that is being inserted, then the function returns false; if it does not match, then it calls `expandMap` at the position.

If a call to `expandMap` fails, then the `failCount` is incremented and the return value is examined. If `failCount` equals `maxFailCount`, then an atomic bit-mark is placed on the contents of `local` at `pos`, and `expandMap` is called. When `expandMap` returns, the thread continues traversal from the `arrayNode` that is guaranteed to be returned (see Sect. 4.3.5). For this situation to arise, the position that

this thread wants to insert into must be highly-contended, so new `arrayNodes` are added until the thread can `insert` without interference from another thread.

The linearization point of this operation, when it returns true, is the CAS on line 13. The same CAS is one of the linearization points when the function returns false, the other two are the atomic reads on lines 8 and 23.

Algorithm 1 insert *key*, *value*

```
1: hash=hashKey(key);
2: local=head;
3: for int r=0; r <keySize−arrayPow;r+=arrayPow do
4:    pos=hash&(arrayLength−1);
5:    hash=hash>>arrayPow;
6:    failCount=0;
7:    node=getNode(local,pos);
8:    while true do
9:       if failCount>maxFailCount then
10:         node=markDataNode(local,pos);
11:      if node==null then
12:         insertThis=allocateNode(value,hash);
13:         if (node=CAS(local[pos],null, insertThis))==null then
14:            watch(null);
15:            return true;
16:         else
17:            free(insertThis);
18:      if isMarked(node) then
19:         node=expandMap(local,pos,r);
20:      if isArrayNode(node) then
21:         local=node;
22:         break;
23:      else
24:         watch(node);
25:         node2=getNode(local,pos)
26:         if node != node2 then
27:            failCount++;
28:            node=node2;
29:            continue;
30:         else if node−>hash == hash then
31:            watch(null);
32:            return false;
33:         else
34:            node=expandMap(local,pos,r);
35:            if isArrayNode(node) then
36:               local=node;
37:               break;
38:            else
39:               failCount++;
40: free(insertThis);
41: watch(null);
42: pos=hash&(arrayLength−1);
43: currValue=local[pos];
44: if currValue == null then
45:    return (CAS(local[pos],null, value)==null);
46: else
47:    return false;
```

4.3.2 Algorithm 2: `Update (key, expectedValue, newValue)`

The `update` function is used to update the value associated with a key that is present in the hash map. This function takes three arguments: the first is the key whose value we would like to update, called `key`; the second is the value that we expect to be associated with this key, called `expectedValue`; and the third is the value that we would like to associate with this key, called `newValue`. The `update` function returns true, if it successfully replaces a `dataNode` whose key and value matches the key and `expectedValue` of this operation. If the key is not present in the hash map, or if the key's associated value does not match `expectedValue`, then the function returns false. In order to reason about the results of a failed CAS operation we require `expectedValue` to be different from `newValue`.

The `update` operation traverses the hash map as described in Sect. 4.2, until it finds a position that is `null`, or that contains a `dataNode`. If a `markedDataNode` is found during the traversal, then `expandMap` is called and the thread continues its traversal. If it is a `dataNode` whose key matches the one being updated, and the value in the `dataNode` matches `expectedValue`, then a CAS is performed which replaces the current `dataNode` with one containing `newValue`.

If the CAS fails, then the return value is examined. If it is a marked version of the node that the CAS attempted to replace, then the thread calls `expandMap` and continues its traversal. If the value returned is an `arrayNode`, then the thread continues its traversal. An arbitrary `dataNode`, `null`, or a `dataNode` whose key and value matches could have been returned as well; the first two indicate that the operation should return false. The return of a `dataNode` whose key and value matches may seem like a successful result; however, it is actually an indication that we may be experiencing the ABA problem. The reasoning is that because we placed the constraint that `expectedValue` may not be equal to `newValue`, then there must have been a state where the key was not present, or the value associated with the key did not match `expectedValue` in order for the CAS to have failed, so we return false in this case. If the traversal is completed without finding a `dataNode` with a key-value pair that matches `key` and `expectedValue`, then the function returns false.

There are several linearization points. Two of these are the atomic reads in the calls to `getNode` at lines 7 and 19; another two of these are the CAS operations at lines 37 and 54. If `update` returns true, then it linearizes upon the return of the appropriate CAS operation. If any of the four lines returns `null` or a pointer to a `dataNode` whose key and value does not match the key and `expectedValue` of this operation, causing `update` to return false, then it is at that point that the operation linearizes. The third point occurs when a failed CAS operation returns a pointer to a `dataNode` whose key and value matches the expected, then the linearization point is between the atomic read in `getNode` and the the completion of the CAS operation. There must have been a state when either the key was not in the map, or the value associated with the key did not match `expectedValue`, and it is at this state that the operation linearizes.

In the worst case, this operation requires `expandMap` to be called until `maxDepth` is reached, at which point it is not possible for there to be any more expansions, by

definition of maxDepth and the constraints on the hash function. Therefore, at this point, the thread will be able to finish its operation with a single CAS or atomic read.

4.3.3 Algorithm 3: `get (key)`

The `get` operation traverses the hash map as described in Sect. 4.2, until it finds a position that is `null`, or that contains a `dataNode`. If it is a `dataNode` whose key matches, then the value associated with the key is returned; otherwise, `null` is returned.

The point at which this operation linearizes is the atomic read in the call to `getNode` (see lines 7 and 17). If a `dataNode` is read, then this thread must announce that it is about to read the node, by calling the `watch` function. If the value changed between the read and the call to `watch`, then the thread retries. If it retries more than `maxFailCount` times, then the thread will mark the address as highly-contended and force an expansion; the number of times that this can occur is equal to `maxDepth`. If `maxDepth` is reached, then the thread can no longer read `dataNodes`, only `null` or values, as such the thread simply returns the value that it reads at this level (see Sect. 4.1).

4.3.4 Algorithm 4: `remove (key, expectedValue)`

The `remove` operation is nearly identical to the `update` operation, it can be treated as a specialized version of `update` where the only difference is that instead of replacing a `dataNode` with another `dataNode`, it replaces it with `null`. It has the same logic for determining when an operation returns true or false, the same bound on the number of loop iterations, and the same linearization points.

4.3.5 Algorithm 5: `expandMap (local, pos, right)`

This function is used to expand the map when there is a hash collision. If the current value at `pos` in `local` is marked, then it is guaranteed that when the function returns, the contents of `pos` in `local` are an `arrayNode` that holds an unmarked version of the node that was there before.

First, `expandMap` reads the current value at `pos`. If it is not an `arrayNode`, then it allocates a new one, calculates the position where the node that was there previously belongs on the`arrayNode`, and sets the pointer at that position equal to the location of the node. Next, it uses a CAS to attempt to replace that node with the arrayNode (see line 10). This function returns the allocated `arrayNode`, if the CAS is successful; otherwise, it returns false.

The atomic read in the call to `getNode` on line 1 is the linearization point, if this operation returns false; the CAS on line 10 is the linearization point, if this operation returns true.

An optimization that we use in the implementation is that if an operation is attempting to `insert` a node that collides with a node that is currently in the map, then the `expandMap` algorithm creates an `arrayNode` or a series of them, that contains both nodes, and then performs the CAS.

Algorithm 2 Update $key, expectedValue, newValue$

```
1: hash=hashKey(key);
2: local=head;
3: result=false;
4: for int r=0; r<keySize−arrayPow;r+=arrayPow do
5:     pos=hash&(arrayLength−1);
6:     hash=hash>>arrayPow;
7:     node=getNode(local,pos);
8:     if isArrayNode(node) then
9:         local=node;
10:    else if isMarked(node) then
11:        local=expandMap(local,pos,r);
12:    else if node==null then
13:        break;
14:    else
15:        watch(node);
16:        if node != getNode(local,pos) then
17:            failCount=0;
18:            while node != getNode(local,pos) do
19:                node=getNode(local,pos);
20:                watch(node);
21:                failCount++;
22:                if failCount>maxFailCount then
23:                    markDataNode(local,pos);
24:                    local=expandMap(local,pos,r);
25:                    break;
26:            if isArrayNode(node) then
27:                local=node;
28:                continue;
29:            else if isMarked(node) then
30:                local=expandMap(local,pos,r);
31:                continue;
32:            else if node==null then
33:                break;
34:        if node−>hash == hash then
35:            if node−>value != expectedValue then
36:                break;
               insertThis=allocateNode(newValue,hash);
37:            if (node2=CAS(local[pos],node,insertThis))==node then
38:                result= true;
39:                break;
40:            else
41:                free(insertThis);
42:                if isArrayNode(node2) then
43:                    local=node2;
44:                else if isMarked(node2)∧unmark(node2)==node then
45:                    local=expandMap(local,pos,r);
46:                else
47:                    break;
48:        else
49:            break;
50: if r >= keySize−arrayPow then
51:     pos=hash&(arrayLength−1);
52:     currValue=local[pos];
53:     if currValue == expectedValue then
54:         result= (CAS(local[pos], expectedValue, newValue) == expectedValue);
55:     else
56:         result=false;
57: else if result then
58:     safeFreeNode(node);
59: watch(null);
60: return result;
```

Algorithm 3 get *key*

```
1: hash=currHash=hashKey(key);
2: local=head;
3: result=null;
4: for int right=0;right<keySize−arrayPow;right+=arrayPow do
5:    pos=hash&(arrayLength−1);
6:    hash=hash>>arrayPow;
7:    node= getNode(local,pos);
8:    if isArrayNode(node) then
9:       local=node;
10:   else if node==null then
11:      break;
12:   else
13:      watch(node);
14:      if node != getNode(local,pos) then
15:         failCount=0;
16:         while node != getNode(local,pos) do
17:            node=getNode(local,pos);
18:            watch(node);
19:            failCount++;
20:            if failCount>maxFailCount then
21:               markDataNode(local,pos);
22:               local=expandMap(local,pos,r);
23:               break;
24:         if isArrayNode(node) then
25:            local=node;
26:            continue;
27:         else if isMarked(node) then
28:            local=expandMap(local,pos,r);
29:            continue;
30:         else if node==null then
31:            break;
32:      if node−>hash == currHash  then
33:         result=node−>value;
34:      break;
35: if r >= keySize−arrayPow then
36:    pos=hash&(arrayLength−1);
37:    result=local[pos];
38: watch(null);
39: return  result;
```

4.4 Memory Management

This section discusses the allocation and reuse of memory. When designing concurrent applications, choosing an appropriate memory management scheme is important, and the one chosen must be thread-safe. As the standard memory allocator is blocking, special provisions must be made for lock-free and wait-free programs. In order for the hash map to behave in a wait-free manner, the user must choose a memory allocator that can manage memory in a wait-free manner [31].

Furthermore, this memory manager must be able to handle the ABA problem [4] correctly, because this problem is fundamental to all CAS-based systems [24]. To prevent the ABA problem we ensure that the values stored in the `dataNode` remain unchanged while any thread is using that `dataNode`. Any update to the value associated with a key is done by replacing the `dataNode` that is associated with that

Algorithm 4 remove $key, expectedValue$

```
1: currHash=hash=hashKey(key);
2: local=head;
3: result=false;
4: for int r=0; r<keySize−arrayPow;r+=arrayPow do
5:    pos=hash&(arrayLength−1);
6:    hash=hash>>arrayPow;
7:    node=getNode(local,pos);
8:    if isArrayNode(node) then
9:       local=node;
10:   else if isMarked(node) then
11:      local=expandMap(local,pos,r);
12:   else if node==null then
13:      break;
14:   else
15:      watch(node);
16:      if node != getNode(local,pos) then
17:         failCount=0;
18:         while node != getNode(local,pos) do
19:            node=getNode(local,pos);
20:            watch(node);
21:            failCount++;
22:            if failCount>maxFailCount then
23:               markDataNode(local,pos);
24:               node=expandMap(local,pos,r);
25:               break;
26:         if isArrayNode(node) then
27:            local=node;
28:            continue;
29:         else if isMarked(node) then
30:            local=expandMap(local,pos,r);
31:            continue;
32:         else if node==null then
33:            break;
34:      if node−>hash == currHash then
35:         if node−>value != expectedValue then
36:            break;
37:         if (node2=CAS(local[pos],node,null))==node then
38:            safeFreeNode(node);
39:            result= true;
40:            break;
41:         else
42:            if isArrayNode(node2) then
43:               local=node2;
44:            else if isMarked(node2)∧unmark(node2)==node then
45:               local=expandMap(local,pos,r);
46:            else
47:               break;
48:      else
49:         break;
50: if r >= keySize−arrayPow then
51:    free(insertThis);
52:    pos=hash&(arrayLength−1);
53:    currValue=local[pos];
54:    if currValue ==expectedValue then
55:       result = (CAS(local[pos], expectedValue, null) == expectedValue);
56:    else
57:       result=false;
58: watch(null);
59: return result;
```

Algorithm 5 expandMap *local*, *pos*, *right*

```
1: node= getNode(local,pos);
2: watch(node);
3: if isArrayNode(node) then
4:    return node;
5: if node !=(node2=getNode(local,pos)) then
6:    return node2;
7: aNode=alloc(sizeof(arrayNode));
8: newPos=(node->hash>>(right+arrayPow))& (arrayLength-1);
9: aNode[newPos]=node;
10: if (node2=CAS(local[pos]), node, aNode)) == node then
11:    return aNode;
12: else
13:    aNode[newPos]=null;
14:    free(aNode);
15:    return node2;
```

key with a new one with the same key. To achieve this we used Michael's ABA-free approach to safe memory-reclamation, called hazard pointers [24].

Hazard pointers work by having each thread announce the address of the memory it is about to access [24]. In our algorithm each thread performs an atomic read at a position on an `arrayNode` and if it is a `dataNode`, the thread writes the address of the `dataNode` to a global array. The thread then checks to ensure that, between reading the `dataNode` and writing to the global array, the node was not removed from that location. If it was removed, then the thread retries; this retrying is what makes some other algorithms that use hazard pointers lock-free. In our algorithm we using the atomic bitmark and expansion to bound the number of times a retry is attempted. In practice retrying rarely occurs. Additionally, since values and not `dataNodes` are stored on the `arrayNodes` located at max depth, there is no need to perform a hazard pointer read at max depth, and the value read can be operated on without concern.

Michael's hazard pointer implementation is wait-free if you can place a reference into the watched address list in a wait-free manner. This consists of reading the contents of an address, storing the value read into the global list, re-reading the contents, and comparing the two values to ensure that they are the same. If they are different it must retry until they are the same. In most algorithms this process is lock-free, because the number of times the algorithm must retry is not bounded. That is not the case in our algorithm, because of how we use atomic bitmarks and the fact that an `arrayNode` cannot be removed. The wait-free property of hazard pointers and the minor adjustments made to implement this algorithm in our code mean that `watch` and `safeFreeNode` are both wait-free (see [24]).

There are several existing approaches to wait-free memory management. An approach that includes wait-free memory allocation and reclamation is found in [31]. For testing purposes we use the Lockless library [21] for lock-free memory allocation, and hazard pointers for wait-free memory reclamation as presented in [24]. To make the entire system wait-free, the user would have to supply their own wait-free memory allocator as the system calls involved in the allocation of memory are beyond the scope of this paper.

4.4.1 Algorithm 6: `watch (value)`

This function uses a thread-local variable, `threadID`, and a global array, `watchedNodes`, to alert other threads of the node a particular thread is using. Watching is done before any read or write operations on the hash map. Each thread has a unique value form 0 to `Threads` as their `threadID`, this corresponds to the position on the `watchedNodes` array where it stores the node that it is about to use. For more information please review Sect. 4.4.

Algorithm 6 watch *value*

```
1: watchedNodes[threadID]=value;
```

4.4.2 Algorithm 7: `safeFreeNode (nodeToFree)`

This function is used to ensure that memory is not freed while another thread is using it. It checks the `watchedNodes` array for the address of `nodeToFree`, and if it is not present, then the node is freed. If it is present, then the `nodePool` (a thread-local linked list that holds pointers to nodes that we want to remove from the map, but cannot because they are in `watchedNodes`) is checked for nodes that are no longer being used, if one is found then that node is freed and this node takes its place in the `nodePool`. Otherwise, additional space is added for this node.

4.4.3 Algorithm 8: `allocateNode (value, hash)`

This function reuses nodes that have been stored in the `nodePool`; if no node is available, then a new node is allocated. The thread first checks its thread-local `nodePool` for a node that is no longer being referenced; if a node is found, then the thread returns a pointer to that node; otherwise, the thread allocates a new node.

4.5 Supporting Functions

This section briefly describes the supporting functions referenced in the pseudocode of the preceding algorithms.

(a) `getNode`: shown as Algorithm 9, returns the pointer held at the specified position, `pos`, on the `arrayNode`, `local`, that is currently being examined.
(b) `isMarked`: shown as Algorithm 10, returns `true` if the pointer has a bitmark at its least significant bit; this reveals a `markedDataNode`.
(c) `isArrayNode`: shown as Algorithm 11, returns `true` if the pointer has a bitmark at its second-least significant bit.
(d) `markDataNode`: shown as Algorithm 12, uses an atomic `OR` operation to place a bitmark on the value held at `pos` on `local`.

Algorithm 7 safeFreeNode *nodeToFree*

```
1: freeable=true;
2: for int i=0; i<Threads; i++ do
3:    if i==threadID then
4:       continue;
5:    else if nodeToFree == watchedNodes[i] then
6:       freeable=false;
7:       break;
8: if freeable then
9:    free(nodeToFree);
10: else
11:    list=nodePool[threadID];
12:    while list != null do
13:       node=list−>value;
14:       freeable=true;
15:       for int i=0; i<Threads; i++ do
16:          if i==threadID then
17:             continue;
18:          else if node == watchedNodes[i] then
19:             freeable=false;
20:             break;
21:       if freeable then
22:          free(list−>value);
23:          list−>value=nodeToFree;
24:          return ;
25:       else
26:          list=list−>next;
27:    pNode=allocate();
28:    pNode−>next = list;
29:    pNode−>value=nodeToFree;
30:    nodePool[threadID]=pNode;
```

(e) `unmark`: shown as Algorithm 13, expects a pointer to a `dataNode` or a `markedDataNode`, and returns a pointer without a mark on the least significant bit.
(f) `free`: a function to free memory using the system's memory management scheme.
(g) `allocate`: a function to allocate memory using the system's memory management scheme.

5 Correctness

In this section we outline a correctness proof. For brevity, we give informal proofs; these follow the style in [23]. Several useful definitions follow. Abbreviations of the form `U11` are used; the letter is the first letter of the corresponding operation e.g. U11 refers to the eleventh line of the `update` algorithm pseudocode.

(1) For all times `t`, a node is in the hash map at `t`, if and only if at `t` it is reachable by following pointers starting from the `head`.
(2) For all times `t`, the state of the hash map is represented as $S_{n,m,p}$ where `n`, `m`, and `p` are defined as follows.
   (a) `n` : the number of `dataNodes` in the hash map at `t`.

Algorithm 8 allocateNode *value*, *hash*

```
1: ppNode=pNode=nodePool[threadID];
2: node = null;
3: while pNode != null do
4:    freeable=true;
5:    for int i=0; i<Threads; i++ do
6:       if i==threadID then
7:          continue;
8:       else if pNode−>value == watchedNodes[i] then
9:          freeable=false;
10:         break;
11:   if freeable then
12:      if ppNode==pNode then
13:         nodePool[threadID]=pNode−>next;
14:      else
15:         ppNode−>next=pNode−>next;
16:      node=pNode−>value;
17:      free(pNode);
18:      break;
19:   else
20:      ppNode=pNode;
21:      pNode=pNode−>next;
22: if node == null then
23:    node=allocate();
24: node−>value=value;
25: node−>hash = hash;
26: return node;
```

Algorithm 9 getNode *local*, *pos*

```
1: res=&local[pos];
2: return res;
```

Algorithm 10 isMarked *node*

```
1: res=(node&0x1);
2: return res;
```

Algorithm 11 isArrayNode *node*

```
1: res=(node&0x2);
2: return res;
```

Algorithm 12 markDataNode *local*, *pos*

```
1: address=&local[pos];
2: res= atomic_OR_and_fetch(address,0x1)
3: return res;
```

Algorithm 13 unmark *node*

```
1: res=(node | 0x1);
2: return res;
```

(b) `m` : the number of `markedDataNodes` in the hash map at `t`.
(c) `a` : the number of `arrayNodes` in the hash map at `t` (this excludes the main array).

For example, the hash map is in state $S_{2,1,0}$ if it contains exactly two `dataNodes`, one `markedDataNode`, and zero `arrayNodes`.

Lemma 1 *The hashed key of a* `dataNode` *never changes while it is in the hash map.*

Lemma 2 *A* `markedDataNode` *is not unmarked until the corresponding expansion has occurred.*

Lemma 3 *An* `arrayNode` *is never removed from the hash map.*

5.1 Safety

To prove safety, we attempt to prove Claim 1.

The hash map is in a valid state, if and only if it matches the definition of some state $S_{n,m,a}$ that is reachable, through the specified transitions, from the initial state $S_{0,0,0}$. The state of the map changes upon the successful execution of any of the following lines: `markDataNode` line 2, `I13`, `R37`, or `E10` (see Sect. 4.3). In Fig. 3, these lines are abbreviated as follows: `markDataNode` line 2 which marks a node becomes `M`, `I13` which inserts a `dataNode` becomes `I`, `R37` which removes a node becomes `R`, and `E10` which unmarks a `markedDataNode` and adds a new `arrayNode` becomes `N`. Transitions that occur on the execution of `markDataNode` line 2 from $S_{1,1,0}$ and $S_{2,1,0}$ have been omitted for clarity.

Claim 1 *All transitions are consistent with the hash map's semantics. If the hash map is in a valid state, then if a CAS succeeds a correct transition occurs, as shown in the state transition diagram in Fig. 3.*

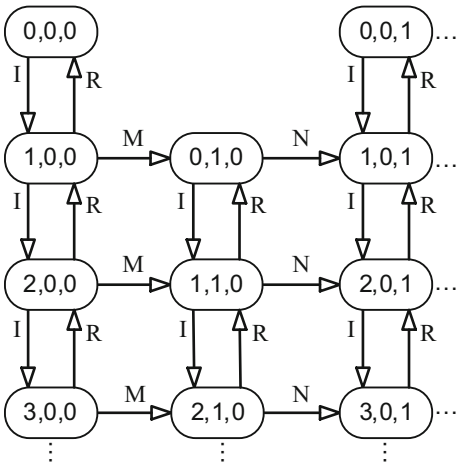


Fig. 3 A state transition diagram for the hash map

In the case of a successful `update` operation the state triple does not change; however, the set of all `dataNodes` that exist in the map is changed (see Sect. 4.3.2). Specifically, a `dataNode` is atomically removed from the set and replaced by a `dataNode` with the same key but a different associated value, this occurs at line U38.

We prove Claim 1 by induction. In the basis step, we assume that the hash map is in the valid, initial state $S_{0,0,0}$. We take Claim 1 to be the induction hypothesis. In the inductive step, we show that, at any time `t`, the application of any transition on a valid state yields a valid state.

Lemma 4 *If successful, the atomic* `OR` *operation in line* `I11` *takes the hash map to a valid state, and marks a* `dataNode`*.*

Lemma 5 *If successful, the CAS on line* `I13` *takes the hash map to a valid state, and inserts a* `dataNode` *into the set.*

Lemma 6 *If successful, the CAS on line* `U38` *does not change the state, and updates the value associated with a key.*

Lemma 7 *If successful, the CAS on line* `R37` *takes the hash map to a valid state, and removes a* `dataNode` *from the set.*

Lemma 8 *If successful, the CAS on line* `E10` *takes the hash map to a valid state and replaces a* `markedDataNode` *with an* `arrayNode` *that contains an unmarked version of the* `markedDataNode`*.*

Theorem 1 *Claim 1 is true at all times.*

5.2 Linearizability

Our hash map is linearizable, because all of its operations have linearization points (see Sect. 4.3 for details).

The linearization points below are presented for each operation, when executed concurrently with any other operation of the hash map. If there is no concurrent execution, then linearizability is not applicable, because the definition of a linearization point is meaningless when defined on a single operation. In the case of a single operation, that of sequential execution, correctness of the algorithms becomes much easier to prove; such proofs are omitted. The linearization points of the supporting algorithms are trivial to prove. Due to the composability of linearizability we do not need to further consider the supporting functions. See Sect. 4.4 for a discussion of the linearizability of the memory management functions.

Lemma 9 *Every* `get` *operation takes effect upon its read on line* `G06`*.*

Lemma 10 *Every* `update` *and* `remove` *operation that returns* `true` *takes effect upon its CAS on lines* `U38` *and* `R37`*, respectively.*

Lemma 11 *Every* `update` *and* `remove` *operation that returns* `false` *takes effect when a* `dataNode` *with a different key is encountered during traversal (see Sect. 4.3.2).*

Lemma 12 *Every* `insert` *operation that returns true takes effect upon its CAS on line* `I13`.

Lemma 13 *Every* `insert` *operation that returns false takes effect upon its CAS on line* `I13`*, its atomic read on line* `I08`*, or its atomic read at line* `I23`.

Given the derived linearization points, we are able to provide a valid sequential history from every concurrent execution of the hash map's operations; this proves Theorem 2.

Theorem 2 *The hash map's operations are linearizable.*

5.3 Wait-Freedom

To prove wait-freedom we must show that every call to `insert`, `update`, `get`, and `remove` returns in a bounded number of steps [19]. This is trivial to prove for the `get`, `update`, and `remove` operations as they are bounded by a for-loop, that runs at most $maxDepth$ times, and the progress of these operations is unhindered by the side effects of any combination of concurrent operations. To prove wait-freedom for `insert` we need to show that the number of operations that may linearize before a particular `insert` operation is bounded [19].

We need only consider those `insert` operations that act on the same position in the hash map, as disjoint operations may proceed in parallel without issue. Furthermore, operations that attempt to `insert` the same key at the same position at the same time do not break the wait-free progress guarantee, because one operation will complete the CAS successfully, and the others will fail and will not retry. However, when concurrent `insert` operations with different keys attempt to work on the same position they would retry infinitely if it were not for `maxFailCount` (see Sect. 4.3.1), which is an upper bound on the number of times that the `insert` operations would conflict before an expansion occurred at that position. In the worst-case, the expansions would be performed until `maxDepth` was reached, with `maxFailCount` attempts at expansion being needed every time.

All of these operations complete in a finite number of steps; this is expressed in Theorem 3. Theorem 4 follows directly from Theorems 1, 2, and 3.

Lemma 14 *The* `insert` *operation completes in a number of steps that is bounded by* `maxDepth` $*$ `maxFailCount`.

Lemma 15 *The* `update` *operation completes in a number of steps that is bounded by* `maxDepth`.

Lemma 16 *The* `get` *operation completes in a number of steps that is bounded by* `maxDepth`.

Lemma 17 *The* `remove` *operation completes in a number of steps that is bounded by* `maxDepth`.

Theorem 3 *All operations of the algorithm are* $\in O(1)$*, in the worst case.*

Theorem 4 *The algorithm is wait-free.*

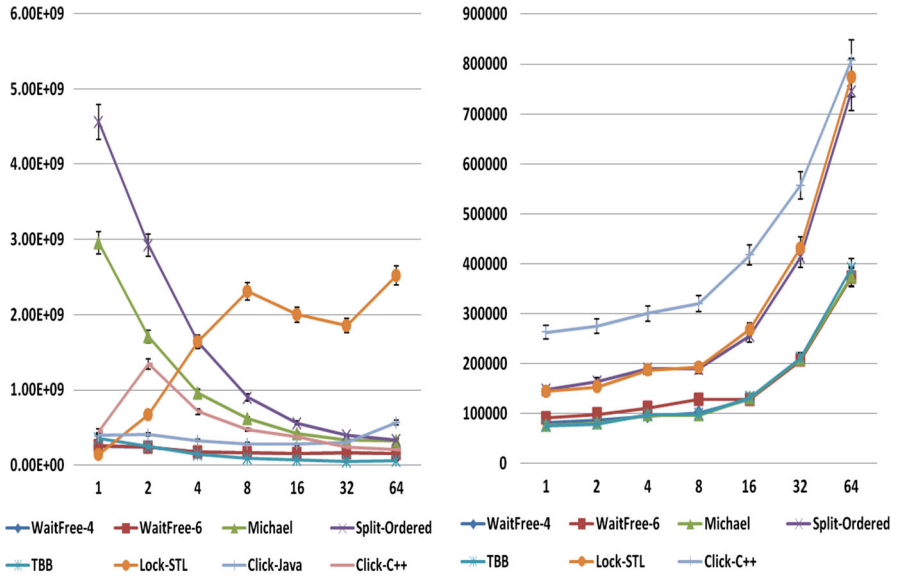
6 Performance Evaluation

We tested several algorithms against our wait-free implementation; we tested with two different values for `arrayLength`, to show the space-time trade-off that this parameter represents. The values that we chose for the `arrayLength` were four (WaitFree-4) and six (WaitFree-6). As there are no other wait-free hash maps in the literature we chose the best available lock-free maps as well as a standard locking algorithm to test against. The locking solution that we include is the C++11 standard template library hash map protected by an optimized global lock (Lock-STL) [16]. The lock-free algorithms, from the literature, that we compare against are Split-Ordered Lists (Split-Ordered) [30] and Michael's lock-free hash map (Michael) [23]. We use the freely available implementations of Split-Ordered Lists and Michael's hash map that are provided by the Concurrent Data Structures library [18].

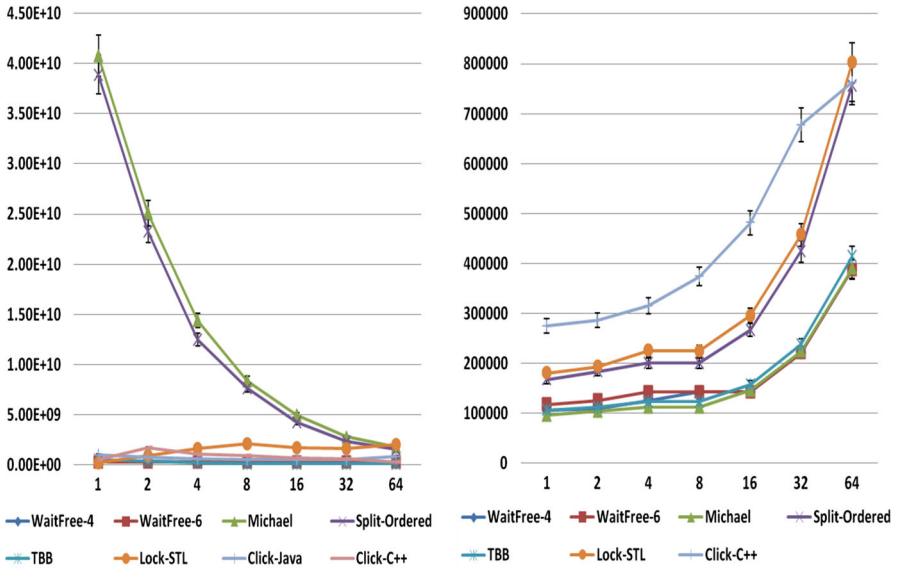
We also compare against two versions of Click's hash map. The first version is provided by him, and is written in Java (Click-Java) [2]. In order to avoid an unfair comparison by comparing C/C++ implementations to Java code, we include the second version which is provided by nbds (Click-C++) [6], and is written in C++. We also compare against Intel TBB's implementation (TBB) [15], because it is known to have high performance.

Careful attention has been paid to the comparability of the different implementations; for example, all tested data structures are able to accept different initial capacities. We only timed the operations of the hash map, avoiding any performance overhead of memory management and any overhead due to the testing itself. All data shown is the average of thirty runs, which were made to minimize the effects of any extraneous factors in the system. All tests were run on a SuperMicro server with four sockets, each populated by a sixteen-core AMD Opteron 6272 processor at 2.1 GHz, and a total of 64 gigabytes of RAM. The machine was running 64-bit Ubuntu Linux version 11.04, and all code was compiled with g++4.7, with level three optimizations enabled. The testing variables for the graph presented in Fig. 4 include creating a hash map that has an initial capacity of 2^{10} elements. This hash map was filled to its capacity and then we performed one million operations.

We divided our operations into three different kinds of distributions. The first type of distribution is based on a reported typical operation mix for hash maps [30]. This mix was reported without mention of an `update` function. We run the reported distribution, 88 % `get`, 10 % `insert`, 0 % `update` 2 % `remove` and a modified version that includes calls to `update`, 88 % `get`, 8 % `insert`, 2 % `update` 2 % `remove`. The second kind of distribution involves inverting the two versions of the aforementioned typical usage distribution within reason by moving the focus from the `get` operation to the `insert` and `update operations`; this yields the following operation mixes:

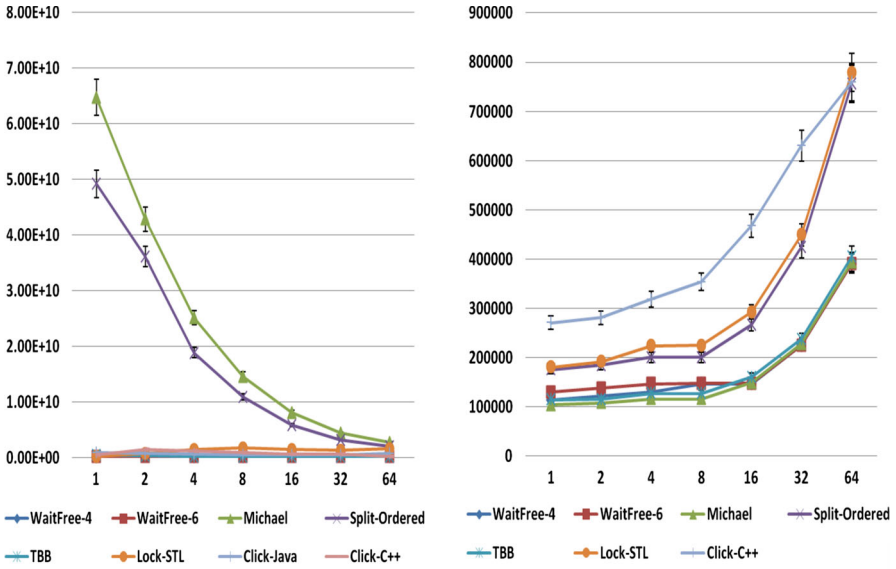


(a)

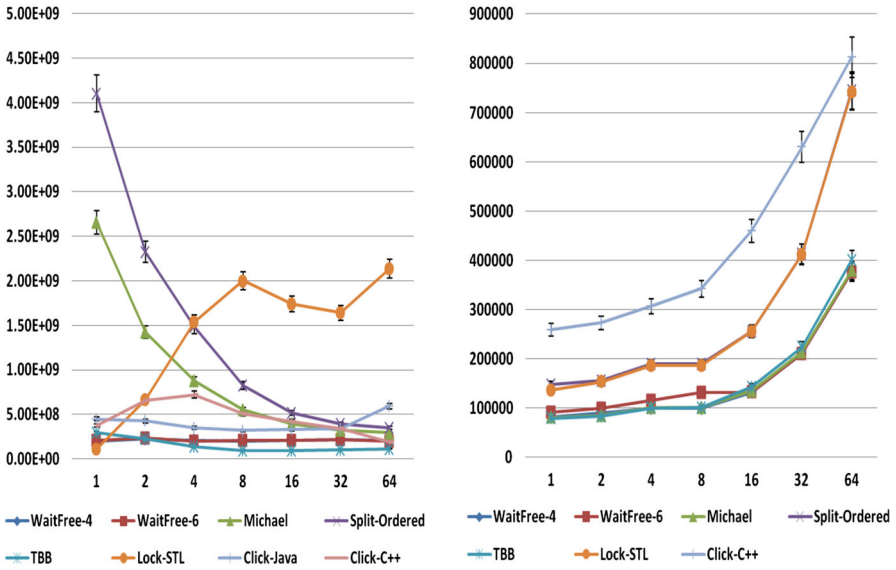


(b)

Fig. 4 Hash map performance results for different operation mixes. **a** 10 % Get, 18 % Insert, 70 % Update, 2 % Remove. **b** 10 % Get, 70 % Insert, 18 % Update, 2 % Remove. **c** 10 % Get, 88 % Insert, 0 % Update, 2 % Remove. **d** 25 % Get, 25 % Insert, 25 % Update, 25 % Remove. **e** 34 % Get, 33 % Insert, 0 % Update, 33 % Remove. **f** 88 % Get, 8 % Insert, 2 % Update, 2 % Remove. **g** 88 % Get, 10 % Insert, 0 % Update, 2 % Remove

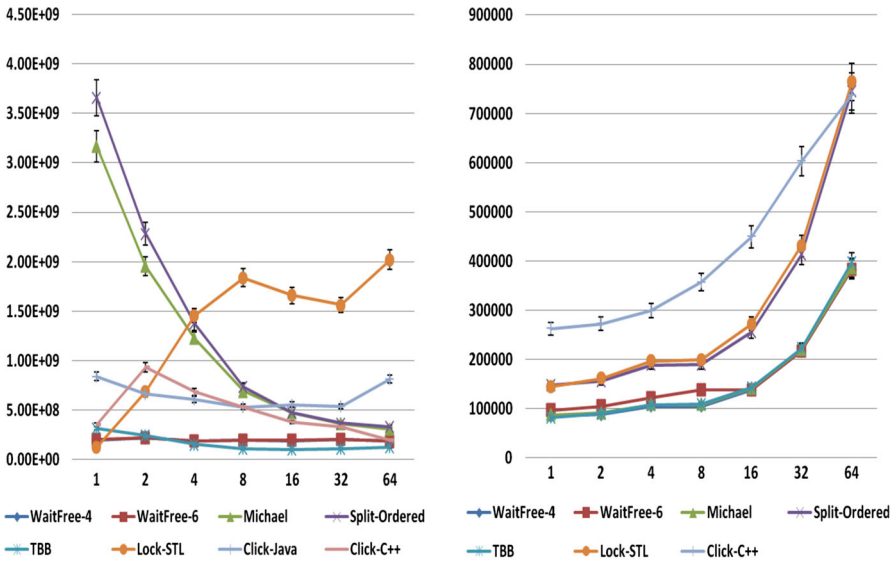


(c)

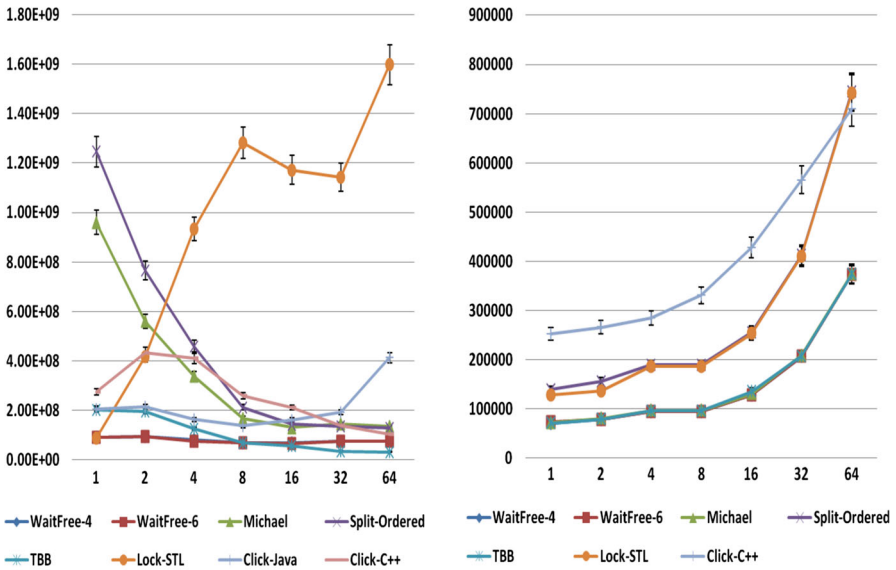


(d)

Fig. 4 continued

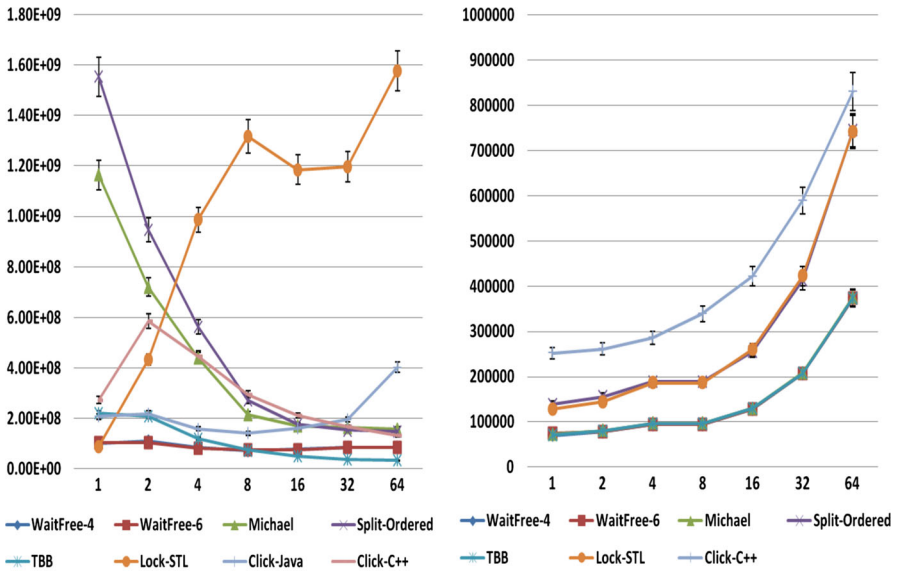


(e)



(f)

Fig. 4 continued



(g)

Fig. 4 continued

10 % `get`, 88 % `insert`, 0 % `update` 2 % `remove`; 10 % `get`, 70 % `insert`, 18 % `update` 2 % `remove`; and 10 % `get`, 18 % `insert`, 70 % `update` 2 % `remove`. The third distribution consists of a more even mix of operations. We have two of these distributions; one includes `update`: 25 % `get`, 25 % `insert`, 25 % `update` 25 % `remove`; one does not include `update`: 34 % `get`, 33 % `insert`, 0 % `update` 33 % `remove`.

The performance results in Fig. 4 show that, on average, our wait-free algorithm outperforms the traditional blocking design by a factor of 7 or more, and it performs faster than the lock-free algorithms typically by a factor of 15. The lack of scalability of the blocking solution is a result of the fact that the lock is applied to all operations, not only those that conflict. Both lock-free solutions scale; however, they perform worse when more `insert` operations are performed, because the `insert` operations trigger more global resizes. Due to the incremental approach that we take to resizing the hash map, we see performance improvements over the other designs in the tested scenarios except for TBB. The other lock-free designs show an average of a 17.5 times performance decrease when compared to Intel's TBB implementation. In contrast, our approach is competitive with only a 14 % loss in performance to provide the stronger progress guarantee of wait-freedom.

On average, the lock-free algorithms use 1.8 times more memory than our algorithm, and the blocking approaches use 1.4 times more memory than our design. When we compare the two different configurations of our algorithm, we see that when we set the `arrayLength` to 6 we use 4 % more memory, but complete the test runs 5 % faster. In general, it is advisable to set the size of the main array equal to the ceiling of the binary logarithm of the expected number of elements; this allows the hash

map to perform a minimal number of resizes, without using too much memory. The `arrayPow` determines how much space is added when a hash collision occurs; it should be set based on the expected number of hash collisions. The `maxFailCount` should be set to the expected number of threads that will compete for a single location in the hash map; in practice, the `failCount` never surpassed 3, but a value of 10 was used for testing. If `maxFailCount` is set too low, then the hash map may be unnecessarily expanded.

The following graphs show the average number of nanoseconds per thread that each operation took to execute the test versus the number of threads, and the average number of kilobytes per thread for each test. These graphs contain error bars which represent a 95 % confidence interval for the results. The memory results for the Java version of Click's hash map were not able to be completely separated from the overhead of the virtual machine; so, these are not reported here.

7 Relevance

We believe that our wait-free hash map allows significant performance increases across any shared-memory parallel architecture. The most pertinent use of our data structure would be in a real-time system where the guarantees of a wait-free algorithm are critical for accurate operation of the system [31]. An example of our hash map in such a system is algorithmic trading. In this case, several threads listen to network updates on stock values that are stored in a hash map by ticker symbol. Due to the rate of change of stock prices, a fast data structure is needed.

Our design could provide speedup to a large number of applications, such as those in the fields of: computational biology[22]; simulation [28,34]; discrete event simulation [17]; and search-indexing [36]. Specifically, our data structure could be used in biological research where both search and computation can involve retrieving and processing vast libraries of information [33]. Additionally, our hash map will be used in the implementation of a popular network performance management software solution provided by SevOne [29].

8 Conclusions and Future Work

We presented a wait-free hash map implementation. Our implementation provides the progress guarantee of wait-freedom with significant performance gains over the tested designs. We discussed the relevance of this work and its applicability in the real-world. To facilitate real-world applications, the code for the algorithms that we discuss here is open source, and is freely available on our website at cse.eecs.ucf.edu.

We are currently developing a project that applies advanced program analysis provided by POET [35] to automatically replace standard, blocking hash maps with our wait-free hash map in real-world applications.

Acknowledgements The authors would like to thank the anonymous reviewers for their detailed and helpful suggestions. This research is funded by the National Science Foundation (NSF) under Grant Numbers 1218100 and DUE-0966249; by the NSF Scholarships in Science, Technology, Engineering, and Mathemat-

ics (S-STEM) program under Award No. 0806931; by the UCF Office of Research and Commercialization; by the Department of Energy; and by Sandia National Laboratories.

References

1. Boost, Boost c++ libraries. Available: http://www.boost.org/. January 2012 [Online]
2. Click, C.: A lock-free hash table. http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf. January (2012)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
4. Dechev, D.: The aba problem in multicore data structures with collaborating operations. In: 2011 7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), October 2011, pp. 158 –167
5. Dechev, D., Pirkelbauer, P., Stroustrup, B.: Lock-free dynamically resizable arrays. In: Shvartsman, M. (ed.) Principles of Distributed Systems, ser. Lecture Notes in Computer Science, vol. 4305, pp. 142–156. Springer, Berlin (2006)
6. Dybnis, J.: nbds. https://code.google.com/p/nbds/. October 2014
7. Fagin, R., Nievergelt, J., Pippenger, N., Strong, H.R.: Extendible hashing—a fast access method for dynamic files. ACM Trans. Database Syst. **4**, 315–344 (1979)
8. Feldman, S., LaBorde, P., Dechev, D.: Concurrent multi-level arrays: wait-free extensible hash maps. In: 13th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII) (2013)
9. Fraser, K.: Practical lock-freedom. In: Kuhn, M. (ed.) Computer Laboratory. Cambridge University Press, Cambridge (2004)
10. Gao, H., Groote, J., Hesselink, W.: Almost wait-free resizable hashtables. In: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, April 2004, p. 50
11. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Proceedings of the 15th International Conference on Distributed Computing, ser. DISC '01, pp. 300–314. Springer, London, UK (2001)
12. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, ser. SPAA '04, pp. 206–215. ACM, New York, NY, USA (2004)
13. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers, New York (2008)
14. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
15. Intel, Intel threading building blocks. http://threadingbuildingblocks.org/. November 2011 [Online]
16. ISO/IEC, Standard for programming language c++. September 2011
17. Janssen, C.L., Adalsteinsson, H., Kenny, J.P.: Using simulation to design extremescale applications and architectures: programming model exploration. SIGMETRICS Perform. Eval. Rev. **38**, 4–8 (2011)
18. Khiszinsky, M.: Concurrent data structures. http://libcds.sourceforge.net/. May 2013
19. Kogan, A., Petrank, E.: Wait-free queues with multiple enqueuers and dequeuers. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, ser. PPoPP '11, pp. 223–234. ACM, New York, NY, USA (2011)
20. Larson, P.-K.: Dynamic hashing. BIT Numer Math **18**, 184–201 (1978). doi:10.1007/BF01931695
21. Lockless Inc., Technical specifications for the lockless inc. memory allocator. http://locklessinc.com/technical_allocator.shtml. December 2011
22. Marçais, G., Kingsford, C.: A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. Bioinformatics **27**(6), 764–770 (2011)
23. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA '02: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 73–82. ACM Press, New York, NY, USA (2002)
24. Michael, M.M.: Hazard pointers: safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (2004)

25. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, ser. PODC '96, pp. 267–275. ACM, New York, NY, USA (1996)
26. Microsoft, System.collections.concurrent namespace. http://msdn.microsoft.com/en-us/library/system.collections.concurrent.aspx. Microsoft, 2011, .NET Framework 4
27. Moir, M., Shavit, N.: Concurrent data structures. In: Mehta, D.P., Sahni, S. (eds.) Handbook of Data Structures and Applications, pp. 47–1–47–30. Chapman and Hall/CRC Press, Boca raton, FL (2007)
28. Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M.A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., Sui, X.: The tao of parallelism in algorithms. SIGPLAN Not. **46**, 12–25 (2011)
29. SevOne, Network performance management. http://www.sevone.com/solutions. June 2012
30. Shalev, O., Shavit N.: Split-ordered lists: lock-free extensible hash tables. In: PODC '03: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, pp. 102–111. ACM Press, New York, NY, USA (2003)
31. Sundell, H.: Wait-free reference counting and memory management. In: Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, April 2005, p. 24b
32. Sundell, H., Tsigas, P.: Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In: OPODIS 2004: Principles of Distributed Systems, 8th International Conference, LNCS, vol. 3544, pp. 240–255 (2005)
33. Trelles, O., Prins, P., Snir, M., Jansen, R.C.: Big data, but are we ready? Nat. Rev. Genet. **12**(3), 224–224 (2011)
34. Williams, J.R., Holmes, D., Tilke, P.: Parallel computation particle methods for multi-phase fluid flow with application oil reservoir characterization. In: Particle-Based Methods, ser. Computational Methods in Applied Sciences. Springer Netherlands, vol. 25, pp. 113–134 (2011)
35. Yi, Q.: POET: a scripting language for applying parameterized source-to-source program transformations. Software Pract. Exp. **42**(6), 675–706 (2012)
36. Zhao, Y., Tang, H., Ye, Y.: RAPSearch2: a fast and memory-efficient protein similarity search tool for next generation sequencing data. Bioinformatics **28**(1), 125–126 (2011)