Concurrent Multi-level Arrays: Wait-free Extensible Hash Maps

Steven Feldman, Pierre LaBorde, Damian Dechev
Department of Electrical Engineering and Computer Science
University of Central Florida, Orlando, FL 32816
feldman@knights.ucf.edu, pierrelaborde@knights.ucf.edu, dechev@eecs.ucf.edu

***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 put, 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. Our design is portable because 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, 5 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 8 or higher.**

I. 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], [26] 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 work was funded by the National Science Foundation (NSF) under Grant Numbers 1218100 and DUE-0966249; by the NSF Scholarships in Science, Technology, Engineering, and Mathematics program under Award No. 0806931; by the UCF Office of Research and Commercialization; by the Department of Energy; and by Sandia National Laboratories.

A. 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 [19]; 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].

978-1-4799-0103-6/13/$31.00 ©2013 IEEE

(c) *Portable*: we only rely on atomic operations available on most modern architectures, such as atomic read, atomic write, and Compare-And-Swap (CAS) [13]; all of these operations are standard in C++11 [16]. This ensures that our implementation can be used on a wide range of multi-processor architectures.
(d) *High performance*: our wait-free hash map design outperforms, by a factor of 8 or more, state of the art non-blocking designs. Our design performs a factor of 5 or greater faster than a standard blocking approach.
(e) *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: Section II briefly introduces the fundamental concepts of non-blocking synchronization, Section III discusses related work, Section IV presents the algorithms of our wait-free hash map design, Section V presents an informal proof of correctness, Section VI offers a discussion of our performance evaluation, Section VII provides an overview of the practical impact of our work, and Section VIII offers conclusions and a discussion of our future work.

II. 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 [4]. 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 [27]. 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.

III. Related Work

Research into the design of non-blocking data structures includes: linked-lists [10], [22]; queues [23], [31], [25]; stacks [11], [25]; hash maps [22], [25], [9]; hash tables [29]; binary search trees [8], and vectors [6].

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 [22], 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 [29] [22]. In [9], 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 [29], 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 [3]; the author now claims lock-freedom. Moreover, the work by Click was not published, so we will not compare to it.

IV. 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.

A. 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 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. 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 arrayNodePow-bit sequences, where arrayNodePow is the binary logarithm of the arrayLength; e.g. $A-B-C-D$, where A is the first arrayNodePow-bit sequence, B is the next arrayNodePow-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 $\log_2 \texttt{arrayLength} * \texttt{currentDepth}$. 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 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.

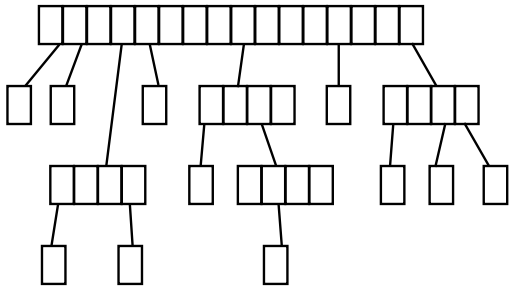


Fig. 1: An illustration of the structure of the hash map.

B. 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.

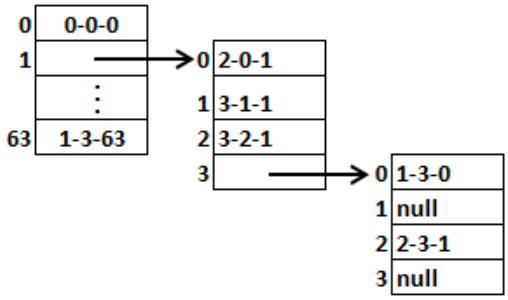


Fig. 2: An example of data stored in the hash map (values not shown).

We discuss the traversal of the hash map using Figure 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 the main arrayNode which has a larger size than every other arrayNode (see Section IV-A). 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 Figure 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 the main memory array. 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.

C. 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 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 default value is ten, though in practice we have not seen a value higher than three. This constant should be set equal to the number of threads, or higher. If maxFailCount is set

lower, performance may be affected because the hash map may be unnecessarily expanded.

If these 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 [22].

1) Algorithm 1 - `put (key, value)`*:* The `put` function is used to insert a key-value pair into the hash map if the key is not already in the hash map and to update the key's value if the key is already in the hash map. A `put` operation traverses the hash map as described in Section IV-B, until it finds a position that is `null`, or contains a `dataNode` with the same key. At this point it performs a CAS, to put the node in the hash map, either at line 19 or line 36, depending on whether `null` or a `dataNode` was initially read; these are two of the four linearization points of the `put` function.

If the CAS fails, then we perform a read of the new value (line 22 or 40 respectively), because the value at that position must have changed in order for our CAS to fail. If this read reveals that the new node has the same key as the one that the `put` operation is trying to insert, then we can reason that this `put` operation completed, and was then immediately overwritten; this means that the reads in lines 22 and 40 are the remaining two linearization points for the `put` function.

However, if the read (line 22 or 40) revealed an `arrayNode`, then the operation would continue its traversal as normal. If the read revealed a `markedDataNode`, then the thread would have to add an `arrayNode` before continuing its traversal. If the operation repeatedly fails the CAS (line 19 or 36) without being able to discover that its operation has been completed and immediately overwritten, then the thread's `failCount` is increased. If this continues until the `failCount` equals `maxFailCount`, then that means that the position that this thread wanted to insert into is highly contended, so new `arrayNodes` are added until the thread can insert without interference from another thread.

In the worst case, this requires that new `arrayNodes` are added until `maxDepth` is reached, at which point it is not possible for there to be any contention caused by a thread that does not involve the same key, because only duplicate keys could hash to the same position at `maxDepth`, due to our use of a perfect hash function. Therefore, at this point, the thread will be able to finish its operation with a single CAS, whether it succeeds or fails, as described above.

2) Algorithm 2 - `get (key)`*:* The `get` operation searches the hash map for a key. If it finds the key (line 10), then it returns the associated value; otherwise, it returns `null`.

3) Algorithm 3 - `remove (key)`*:* The `remove` operation traverses the hash map until a `dataNode`, `markedDataNode`, or `null` is found. If anything is found, other than a `dataNode`, or a `markedDataNode` with a matching key, then the operation returns `false`, because the key was not found in the hash map; the linearization point is the read where we discovered this (line 6). If a `dataNode` is found, then a CAS is performed to remove it. If the CAS is successful, then the operation returns `true`, and the linearization point is line 13.

Algorithm 1 put *key*, *value*

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

Algorithm 2 get *key*

```
 1: hash=hashKey(key);
 2: local=head;
 3: for int right=0; right< keySize;right+=arrayNodePow do
 4:     pos=hash&(arrayLength-1);
 5:     hash=hash >> arrayNodePow;
 6:     node= unmark(getNode(local,pos));
 7:     if isArrayNode(node) then
 8:         local=node;
 9:     else
10:         if node-> hash == hash then
11:             return node-> value;
12:         else
13:             return null;
```

If the CAS fails, then the position is read again, because the contents of that position must have changed since the last read. If this read reveals an `arrayNode`, then the appropriate position on the `arrayNode` is examined. If the position holds a `markedDataNode`, then the new `arrayNode` is created and examined. However, if the contents of that position are anything else, then we return `true`.

If the position is `null`, then that means a concurrent `remove` operation has already removed this key. If the position is a `dataNode`, or a `markedDataNode`, then this indicates that our `remove` operation was executed concurrently with a `put` operation that took effect immediately after our `remove`, and the read at line 18 is the linearization point.

Algorithm 3 remove *key*

```
 1: hash=hashKey(key);
 2: local=head;
 3: for int R=0; R< keySize;R+=arrayNodePow do
 4:     pos=hash&(arrayLength-1);
 5:     hash=hash >> arrayNodePow;
 6:     node= getNode(local,pos)
 7:     if node == null then
 8:         return false;
 9:     else if isMarked(node) then
10:         local=expandTable(local,pos,node,R);
11:     else if !isArrayNode(node) then
12:         if node-> hash == hash then
13:             if CAS(local[pos], node, null) then
14:                 free(node);
15:                 return true;
16:             else
17:                 node2=getNode(local,pos);
18:                 if isMarked(node2)∧unmark(node2)==node then
19:                     local=expandTable(local,pos,node,R);
20:                 else if isArrayNode(node2) then
21:                     continue;
22:                 else
23:                     return true;
24:         else
25:             return false;
26:     else
27:         local=node;
```

D. Supporting Functions

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

(a) `allocateNode`: a simple function to allocate a node using a wait-free memory management scheme (see Section IV-E).

(b) `expandTable`: adds a new `arrayNode` when there is a hash collision, or a high amount of contention on a single memory location which is indicated by the presence of a `markedDataNode`. This operation uses at most a number of CAS operations equal to the number of threads. No other thread will attempt to work on a `markedDataNode`, except to attempt to perform the expansion itself.

(c) `free`: a function to free memory using a wait-free memory management scheme (see Section IV-E).

(d) `getNode`: returns the pointer held at the specified position, `pos`, on the `arrayNode`, `local`, that is currently being examined.

(e) `isMarked`: returns `true` if the pointer has a bitmark at its least significant bit; this reveals a `markedDataNode`.

(f) `isArrayNode`: returns `true` if the pointer has a bitmark at its second-least significant bit.

(g) `markDataNode`: uses an atomic `and` operation to place a bitmark on the value held at `pos` on `local`.

(h) `unmark`: expects a pointer to a `dataNode` or a `markedDataNode`, and returns a pointer without a mark on the least significant bit.

E. 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 [30].

Furthermore, this memory manager must be able to handle the ABA problem [5] correctly, because this problem is fundamental to all CAS-based systems [24]. We allow the user to choose which memory management scheme they use with our hash map. This adds the benefit of allowing our hash map to use hardware-specific memory management schemes.

There are several existing approaches to wait-free memory management. One approach for wait-free memory reclamation is hazard pointers [24]. An approach that includes wait-free memory allocation and reclamation is found in [30]. For testing purposes we use the Lockless library [20] for lock-free memory allocation, and a custom approach to memory management that is a mix of the techniques developed in [12] and [24]; the details of correct memory management and allocation in wait-free data structures is beyond the scope of this paper.

V. Correctness

In this section we outline a correctness proof. For brevity, we give informal proofs; these follow the style in [22]. Several useful definitions follow. Abbreviations of the form `P10` are used; the letter is the first letter of the corresponding operation e.g. `P10` refers to the tenth line of the `put` 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`.
   (b) `m` : the number of `markedDataNodes` in the hash map at `t`.
   (c) `p` : the number of `arrayNodes` in the hash map at `t` (this excludes the main array).

(3) `Unmark` represents any one of the following lines: `P16`, `P27`, `P45`, `P51`, `R10`, `R19`.

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 node never changes while the node 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.*

A. Safety

To prove safety, we attempt to prove Claim 1.

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 Figure 3.*

Transitions that do not change the state, transitions that could occur on the execution of line `P10` from $S_{1,1,0}$ and $S_{2,1,0}$, and the possible transitions that could occur upon execution of line `P51`, have been omitted for clarity; `P10` marks a node, and `P51` adds a new arrayNode.

The hash map is in a valid state, if and only if it matches the definition of some state $S_{n,m,p}$ that is reachable, through the specified transitions, from the initial state $S_{0,0,0}$. The state of the hash map changes upon the successful execution of the operations in any one of the following lines: `P10, P19, P16, P27, P45, P51, R10, R19, R13, R18` (see Section IV-C).

Operations that fail a CAS operation may still be considered to have linearized. This allows the concurrent execution of a `put` and a `remove`, on the same key, to both complete execution after the successful completion of only one of the operations; the other operation will not be retried, because it has been canceled out in a manner similar to the elimination scheme described in [11]. The operations that fail a CAS, but still linearize do not cause changes in the state, the other thread's successful CAS causes a change in state, except in one case.

In the case that a `put` causes a `remove` operation to fail, and the `remove` is considered to have occurred, the state changes twice. First, from $S_{n,m,p}$ to $S_{n+1,m,p}$, because of the successful `put`. Then, the state changes from $S_{n+1,m,p}$ to $S_{n,m,p}$, because of the `remove`. This second state change is accomplished using the transition that is labeled `R13, R18` — specifically, the line that represents this case, `R18`. This second change of state is the only time that the state changes without a corresponding CAS being executed.

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* `and` *operation in line* `P10` *takes the hash map to a valid state, and marks a* `dataNode`.

Lemma 5. *If successful, the CAS in line* `P19` *takes the hash map to a valid state, and inserts a* `dataNode` *into the set.*

Lemma 6. *If successful, the CAS in line* `P37` *does not change the state, and updates the value associated with a key.*

Lemma 7. *If successful, the CAS in line* `P51` *takes the hash map to a valid state, and adds an* `arrayNode`.

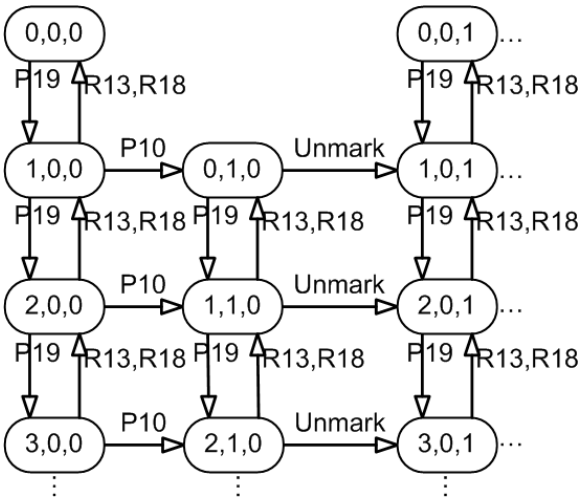


Fig. 3: A state transition diagram for the hash map.

Lemma 8. *If successful, the CAS in line* `R13` *takes the hash map to a valid state, and removes a* `dataNode` *from the set.*

Lemma 9. *If successful, the CAS on any of the lines in* `Unmark` *takes the hash map to a valid state and replaces a* `markedDataNode` *with an* `arrayNode` *that contains an unmarked version of the original* `markedDataNode`.

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

B. Linearizability

Our hash map is linearizable, because all of its operations have linearization points (see IV-C 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.

Lemma 10. *Every* `get` *operation takes effect upon its read on line* `G6`.

Lemma 11. *Every* `remove` *operation that returns* `true` *takes effect upon its CAS on line* `R13`*, or its read on line* `R18` *(see Section IV-C).*

Lemma 12. *Every* `remove` *operation that returns* `false` *takes effect upon its read on line* `R6`.

Lemma 13. *Every* `put` *operation takes effect upon its CAS on line* `P19` *or* `P37`*, or its read on line* `P22` *or* `P40` *(see Section IV-C).*

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.*

C. Wait-Freedom

To prove wait-freedom we must show that every call to `put`, `get`, and `remove` returns in a bounded number of steps [18]. This is trivial to prove for the `get` operation as it is bounded by

a for-loop, that runs at most *maxDepth* times, and its progress is unhindered by the side effects of other operations. To prove wait-freedom for `put` and `remove` we need to show that the number of operations that may linearize before a particular operation is bounded [18].

We need only consider those operations that act on the same position in the hash map, as the progress of disjoint operations can only be hindered by expansions, where the maximum number of expansions for any position is equal to $maxDepth - currentDepth$; in the worst case `maxFailCount` will be reached upon every expansion. This means that the bounds of the operations that modify the hash map, `put` and `remove`, are equal to $(maxDepth - currentDepth) * maxFailCount$, which are all constants.

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* `get` *operation completes in a number of steps equal to* `maxDepth`.

Lemma 15. *The* `put` *operation completes in a number of steps equal to* $(maxDepth - currentDepth) * maxFailCount$.

Lemma 16. *The* `remove` *operation completes in a number of steps equal to* $(maxDepth - currentDepth) * maxFailCount$.

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

Theorem 4. *The algorithm is wait-free.*

VI. 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 five (WaitFree-5). As there are no other wait-free hash maps in the literature we chose the best available lock-free tables as well as a standard locking algorithm to test against. The lock-free algorithms that we compare against are Split-Ordered Lists (Split-Ordered) [29] and Michael's lock-free hash map (Michael) [22]. The locking solution that we include is the C++11 standard template library hash map protected by an optimized global lock (Lock-STL) [16].

Careful attention has been paid to the comparability of the different implementations; for example, all hash maps 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 five runs, which were made to minimize the effects of any extraneous factors in the system. All tests were run on an HP Z600 workstation, with an Intel X5670 hex-core processor running at 2.93 GHz (with Turbo Boost disabled), and six gigabytes of RAM. The machine was running 64-bit Ubuntu Linux version 11.04, and all code was compiled with g++, with optimizations enabled.

The testing variables for the graph presented in Figures 4 and 5 include creating a hash map that has an initial capacity of 2^{18} = 262,144 elements. This number of elements is used because it is the closest power of two to five percent of the expected number of inserted elements in the test case that represent the typical usage scenario; thus, testing the ability of each hash map to resize to accommodate new elements.

This hash map was filled to one percent of its capacity and then 50,000,000 operations were performed. The Boost random number generator [1] was used to avoid the locking version in the standard C++ implementation, while generating test cases concurrently. Using these numbers, each thread was given a stack that included the function calls and operands that were generated.

We selected three different distributions of operations; the first distribution contained 88% `get`, 10% `put`, and 2% `remove` operations (see Fig. 4a). This distribution was selected because it was reported to be typical for use of this data structure [29]. The other distributions that were tested represent the inversion of this test (10% `get`, 88% `put`, 2% `remove`, see Fig. 4c), and a more even distribution (34% `get`, 33% `put`, 33% `remove`, see Fig. 4b). Memory tests were performed for each of these distributions, and the results can be seen in Fig. 5a, 5c, 5b.

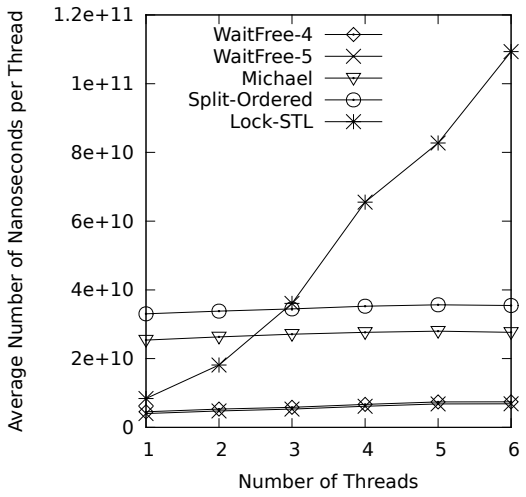
The performance results in Figures 4 and 5 show that, on average, our wait-free algorithm outperforms the traditional blocking design by a factor of 5 or more, and it performs faster than the lock-free algorithms by a factor greater than 8. The lack of scalability of the blocking solution is a result 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 `put` operations are performed, because the `put` 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.

On average, the lock-free algorithms use 11% less memory than our algorithm, and the blocking approach uses 4% less memory than our design. The increase in memory usage that our design demonstrated is explained by the `arrayLength` being set higher to improve performance by presenting more open positions to a thread performing a `put` operation.

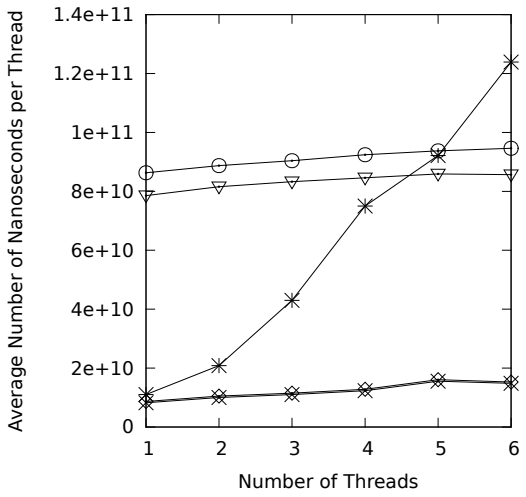
VII. Relevance

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 [30]. 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.

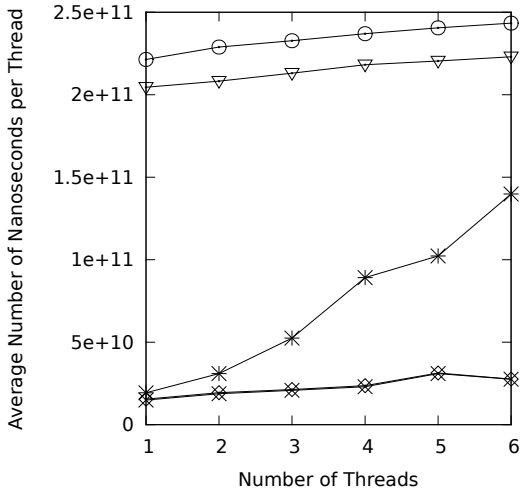
Our design could provide speedup to a large number of applications, such as those in the fields of: computational biology[21]; simulation [27], [33]; discrete event simulation [17]; and search-indexing [35]. Specifically, our data structure could be used in biological research where both search and computation can involve retrieving and processing



(a) 88% Get, 10% Put, 2% Remove

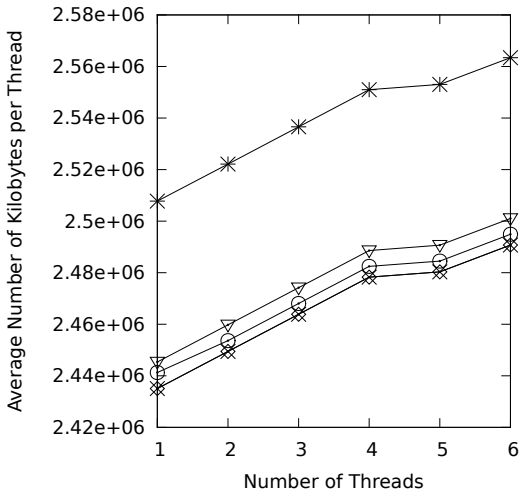


(b) 34% Get, 33% Put, 33% Remove

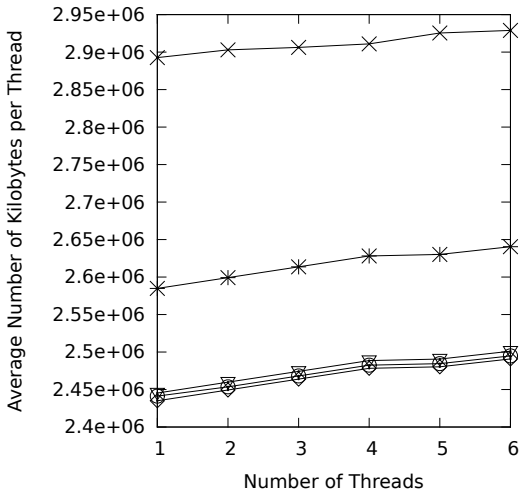


(c) 10% Get, 88% Put, 2% Remove

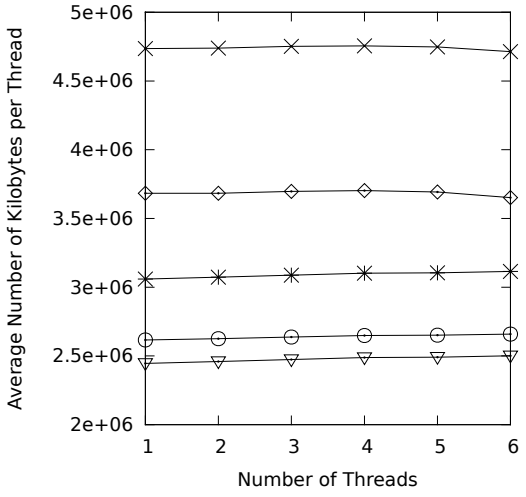
Fig. 4: Average Execution Time in Nanoseconds per Thread



(a) 88% Get, 10% Put, 2% Remove



(b) 34% Get, 33% Put, 33% Remove



(c) 10% Get, 88% Put, 2% Remove

Fig. 5: Average Number of Kilobytes per Thread

vast libraries of information [32]. Additionally, our hash map will be used in the implementation of a popular network performance management software solution provided by SevOne [28].

VIII. 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 we intend to make it freely available on our website at *cse.eecs.ucf.edu*; at present, the code is available under a BSD license upon email request.

We are currently developing a project that applies advanced program analysis provided by POET [34] to automatically replace standard, blocking hash maps with our wait-free hash map in real-world applications and a number of benchmarks such as PARSEC [2].

Acknowledgments

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 Mathematics (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 c++ libraries," http://www.boost.org/, January 2012. [Online]. Available: http://www.boost.org/

[2] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[3] C. Click, "A lock-free hash table," http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf, January 2012.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[5] D. Dechev, "The aba problem in multicore data structures with collaborating operations," in *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2011 7th International Conference on*, oct. 2011, pp. 158 –167.

[6] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Lock-free dynamically resizable arrays," in *Principles of Distributed Systems*, ser. Lecture Notes in Computer Science, M. Shvartsman, Ed. Springer Berlin / Heidelberg, 2006, vol. 4305, pp. 142–156.

[7] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing - a fast access method for dynamic files," *ACM Trans. Database Syst.*, vol. 4, pp. 315–344, September 1979. [Online]. Available: http://doi.acm.org/10.1145/320083.320092

[8] K. Fraser, "Practical lock-freedom," in *Computer Laboratory, Cambridge Univ*, 2004.

[9] H. Gao, J. Groote, and W. Hesselink, "Almost wait-free resizable hashtables," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, april 2004, p. 50.

[10] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC '01. London, UK, UK: Springer-Verlag, 2001, pp. 300–314.

[11] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," in *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '04. New York, NY, USA: ACM, 2004, pp. 206–215.

[12] M. Herlihy, V. Luchangco, P. Martin, and M. Moir, "Nonblocking memory management support for dynamic-sized data structures," *ACM Trans. Comput. Syst.*, vol. 23, pp. 146–196, May 2005.

[13] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. New York, NY, USA: Morgan Kaufmann Publishers, 2008.

[14] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[15] Intel, "Intel Threading Building Blocks," http://threadingbuildingblocks.org/, November 2011. [Online]. Available: http://threadingbuildingblocks.org/

[16] ISO/IEC, "Standard for programming language c++, september 2011," September 2011.

[17] C. L. Janssen, H. Adalsteinsson, and J. P. Kenny, "Using simulation to design extremescale applications and architectures: programming model exploration," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 4–8, March 2011.

[18] A. Kogan and E. Petrank, "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. New York, NY, USA: ACM, 2011, pp. 223–234.

[19] P.-k. Larson, "Dynamic hashing," *BIT Numerical Mathematics*, vol. 18, pp. 184–201, 1978, 10.1007/BF01931695.

[20] Lockless Inc., "Technical specifications for the lockless inc. memory allocator," http://locklessinc.com/technical_allocator.shtml, December 2011.

[21] G. Marçais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, vol. 27, no. 6, pp. 764–770, 2011.

[22] M. M. Michael, "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*. New York, NY, USA: ACM Press, 2002, pp. 73–82.

[23] ——, "Cas-based lock-free algorithm for shared deques," in *Euro-Par '03*, vol. 2790. LNCS, 2003, pp. 651–660.

[24] ——, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, 2004.

[25] Microsoft, "System.collections.concurrent namespace," http://msdn.microsoft.com/en-us/library/system.collections.concurrent.aspx, Microsoft, 2011, .NET Framework 4.

[26] M. Moir and N. Shavit, *Handbook of Data Structures and Applications*. Chapman and Hall/CRC Press, 2007, ch. Concurrent Data Structures, pp. 47–1–47–30.

[27] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," *SIGPLAN Not.*, vol. 46, pp. 12–25, June 2011.

[28] SevOne, "Network performance management," http://www.sevone.com/solutions, June 2012.

[29] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," in *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 2003, pp. 102–111.

[30] H. Sundell, "Wait-free reference counting and memory management," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, april 2005, p. 24b.

[31] H. Sundell and P. Tsigas, "Lock-free and practical doubly linked list-based deques using single-word compare-and-swap," in *OPODIS 2004: Principles of Distributed Systems, 8th Int. Conf., LNCS, volume 3544*, 2005, pp. 240–255.

[32] O. Trelles, P. Prins, M. Snir, and R. C. Jansen, "Big data, but are we ready?" *Nature Reviews Genetics*, vol. 12, no. 3, pp. 224–224, March 2011.

[33] J. R. Williams, D. Holmes, and P. Tilke, "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, 2011, vol. 25, pp. 113–134.

[34] Q. Yi, "Poet: A scripting language for applying parameterized source-to-source program transformations," *Software: Practice and Experience*, 2011.

[35] Y. Zhao, H. Tang, and Y. Ye, "Rapsearch2: A fast and memory-efficient protein similarity search tool for next generation sequencing data," *Bioinformatics*, 2011.