SAND2013-7759C

A Practical Wait-Free Multi-Word Compare-and-Swap Operation

Steven Feldman
University of Central Florida
Feldman@knights.ucf.edu Pierre LaBorde
University of Central Florida
Damian Dechev
University of Central Florida
Sandia National Laboratories
dechev@eecs.ucf.edu

***Abstract*—Algorithms designed for current and future multi-core systems, which are expected to experience an increase of the number of cores by 100x over the next decade, must exhibit strong scaling. The guarantee of progress provided by wait-free algorithms and the fine-grained synchronization methods used in their designs, make them desirable for achieving this goal. However, the design and development of advanced wait-free algorithms is often inhibited by the limitations of portable atomic hardware operations. Typically these operations can manipulate a single address at a time, where many concurrent algorithms need to perform a series of operations on multiple addresses, requiring more advanced synchronization mechanisms such as a wait-free Multi-Word-Compare-and-Swap (MCAS).**

In this paper, we present the first practical MCAS design that is wait-free in all scenarios. This property holds even if interrupts consistently cause a thread to retry a portion of its operation. Our approach uses a progress assurance scheme that allows a blocked thread to announce that it is unable to make progress. In this scenario, other threads would help complete the blocked thread's operation. To ensure that the other threads help complete the delayed operation, each thread incrementally checks for announcements before beginning its own operation. This differs from traditional lock-free helping techniques, where a thread will only help complete an operation that is in conflict with its operation. To support this progress assurance scheme, we designed a novel ABA prevention mechanism that ensures that when multiple threads attempt to execute the same operation, only one thread will be successful. Our design is practical in that it is built from only portable atomic operations (e.g. atomic reads, atomic writes, compare-and-swap), it is efficient in its utilization of memory (i.e. requiring only a single bit to be reserved from each word, not requiring use of explicit memory barriers, and requiring only four words per address in the operation), and has a wait-free progress guarantee.

In a scenario with high contention, with 64 threads executing updates on a single multi-word object, our wait-free design performs, on average, 77.1% more operations than other practical approaches. Over all tested scenarios, our design performs an average 8.3% more operations.

I. Introduction

On-chip parallelism is expected to be the primary area of parallelism growth in future multiprocessor systems [18]. For application developers, meeting the design challenges of current and future multi-core systems demands rethinking fundamental concepts such as how shared data is acted upon and manipulated. Specifically, to cope with the expected limitations of available memory and bandwidth, new algorithms will have to exploit more parallelism within the computation performed on a single datum (i.e. strong scaling [1]).

The development and use of effective shared memory synchronization is pivotal for overcoming serialization bottlenecks and reaching necessary degrees of strong scaling. Concurrent algorithms that are based on mutual exclusion suffer from performance and safety problems in multiprocessor systems. For example, mutual exclusion can restrict the amount of parallelism an algorithm can achieve and lead to hazards such as deadlock, livelock, and starvation.

Non-blocking designs avoid mutual exclusion, and instead focus on increasing the work performed with a single datum. These designs rely solely on hardware atomic primitives, such as compare-and-swap, to increase the amount of work expressed in a single operation [8]. Wait-freedom is a property of non-blocking designs provide a guarantee that each thread makes progress, freeing them from all three aforementioned hazards of mutual exclusion. This differs from lock-free algorithms which are still suitable to thread starvation. Because of this, wait-free algorithms promise to achieve the necessary degree of strong scaling.

Recent research has provided a number of wait-free data structures built from portable hardware-supported operations including hash maps [19], linked lists [21], queues [13], and et al. These data structures often depend

[1]Strong scaling is the scenario when the total problem size stays fixed while the number of processing elements are increased. The challenge is how to synchronize the work of the processing elements in a correct and efficient manner without "wasting" too many cycles on parallelism overhead. In weak scaling, the problem size assigned to each processing element remains constant while the total problem size may increase. In this case, the main challenge is how to add new processing elements to the existing system.

on atomic primitives, such as atomic read, atomic store, and Compare-and-Swap (CAS)[2] to achieve fine-grained shared-memory synchronization in their design.

Unfortunately, these atomic primitives are typically limited to operating on a single address. Advanced algorithms often operate on a series of addresses at a time. For these algorithms, a practical software Multi-Word-Compare-and-Swap (MCAS) operation is a necessity. A wait-free, ABA-free MCAS operations allows a developer to express the semantics of these advanced algorithms without the underlaying MCAS algorithm reducing the progress or safety guarantee of the algorithm.

MCAS is a programming abstraction that allows a thread to update a series of memory addresses in a single step [7]. This update is successful only if the values at these addresses have not changed between the reading of those values and the call to MCAS. A number of recent multiprocessor algorithms and data structures rely on the availability of an efficient software MCAS implementation. The use of MCAS within those algorithms varies greatly, common uses described in literature include:

- A non-blocking hash table implementation [16], which requires an MCAS algorithm to support the use of multi-word length keys and values. This design exhibits improved data locality compared to other designs that access keys and values through references, which can be located on different cache lines.
- A lock-free, array-based priority queue implementation [12], which requires an MCAS to swap a lower priority value with a higher priority value. The authors' methodology ensures that as newer values are pushed to the bottom, older values are pushed to the top.
- A binary search tree implementation [7], that uses MCAS to ensure that concurrent modifications maintain the balanced nature of the tree. Specifically, when removing or adding an element that requires elements to be rotated, the MCAS operation is able to perform all steps of this rotation atomically.
- It has been proposed that for systems with hardware transaction memory support (HTM), a software-based MCAS algorithm can be used instead of the HTM when an operation exceeds the HTM's supported size. This is because an MCAS, can operate on an arbitrary number of memory locations, while most HTM proposals limit the number of locations [17].

This paper presents the first MCAS design that is wait-free and ABA-free in all scenarios of execution. It is built from only portable atomic operations, and performs, on average, more operations per second than other approaches. It performs, on average, 67.8% more operations per second in tests with 64 executing threads.

Our design implements a strategy that replaces the value at each address in an MCAS operation with a descriptor object [3] which can only be removed once the MCAS operation is completed. A thread that reads a descriptor object may choose to help complete the MCAS operation in progress or perform a read-through to return a value.

The key contributions of this work are:

- Wait-free progress: we present the first software MCAS implementation built from portable atomic instructions that ensures wait-free execution in all scenarios. This differs from other designs where helping may result in thread starvation.
- Performance: Provides fast execution in scenarios of high contention; in synthetic tests performed with 64 threads on a 64 core workstation, our design completes, on average, 71.8% more 2-word MCAS operations and 82.1% more 32-word MCAS operations then other designs. On average, over all tested scenarios our design provides 8.6% improved performance compared with other designs.
- Our MCAS operation incorporates a progress assurance scheme that guarantees a thread will make progress.
- Correctness and ABA-freedom: our association between descriptors and MCAS operations allows us to detect when ABA occurs and to prevent it from causing undefined behavior.
- Composable with algorithms that require one of the two least significant bits of the memory word. In contrast, Harris et al. and similar designs that require two bits to be reserved, our design requires a single bit.

A. Road Map

The remainder of this paper is structured as follows: Section II describes other MCAS implementations. Section III and IV provide detailed descriptions of how the algorithm is implemented. Section V provides an informal proof that our model behaves properly in all cases and that of our approach meets all claims made. In Section VI we present experimental data that show how different implementations compare in different use case scenarios. We conclude in Section VIII.

II. Related Work

Israeli et al. present a lock-free and disjoint-access parallel MCAS algorithm [10]. This algorithm requires a thread identifier to be stored alongside the value of a memory address, limiting the number of bits available to the value. This thread identifier is used to access a set of global variables that contain information about the operations that are currently executing in the system. This design does not support the ability to perform a read through to get the current value for that address, but rather requires a thread to help complete any pending operations

[2] An operation with infinite consensus number in the wait-free/lock-free hierarchy

[3] An object that allows an interrupting thread to help an interrupted thread to complete successfully [4].

before proceeding with its own operation. This algorithm is dependent on the LL/VL/SC primitive[4], which is not provided by any contemporary system.

Anderson et al. demonstrate a wait-free MCAS algorithm that is disjoint-access parallel and supports read through parallelism [2]. In contrast to [10], their design requires that each memory word that contains a value is followed by an additional memory word containing auxiliary information about any pending concurrent operations. This information includes the identification of the thread performing a concurrent operation at the address and the information needed to help complete the operation. Using a non-redundant helping scheme, this design chooses not to perform recursion to help complete a conflicting operation, but rather it causes the conflicting operation to be restarted. Like [10] this design requires the LL/VL/SC primitive. A simplified lock-free version of this algorithm was presented by Moir [15]. Attiya et al. [3] have also presented improvements upon this design.

Harris et al. [6] present a lock-free MCAS algorithm that is disjoint-access parallel, supports read through parallelism, and does not depend on LL/VL/SC. Rather this design uses a CAS operation to replace the expected value at an address with a reference to a descriptor object. This design reserves the two lowermost bits of each address to distinguish between values and descriptor objects. To ensure correct behavior of the MCAS algorithm and prevent the ABA problem, Harris et al. designed a "double compare single swap" algorithm. Compared with [10] and [2] their design shows a significant increase in performance and portability.

Sundell [20] proposes a wait-free MCAS algorithm based on a greedy helping scheme. Like [6], his design is disjoint-access parallel, supports read through parallelism, and does not depend on LL/VL/SC. In the first phase of the greedy helping scheme, the thread attempts to place a reference to its MCAS operation's descriptor object at as many of the addresses in its operation as it can. In the next phase, if another MCAS operation holds some of these addresses needed for this operation, then one of the two operations will steal addresses from the other. Unlike [6], Sundell makes no claim that his algorithm is ABA-free, and when examined his algorithm can exhibit undefined behaviors in certain cases caused by the ABA-problem[5].

In Sundell's algorithm the value returned by a failed CAS operation could lead to the thread reattempting the CAS operation. It is possible for a CAS operation to consistently return a value indicating that a thread should reattempt the CAS operation, in this case the algorithm is lock-free and not wait-free.

III. Structures

This section describes the global variables, thread local variables, and descriptor objects we use in our MCAS algorithm. The descriptor objects contain the information necessary to allow a thread operating on an address held by an MCAS operation to determine the logical value of that address and, if necessary, help complete the MCAS operation.

- `CasRow` or Compare-and-Swap Row, is a structure that holds the following word-length values in the following order: address to be operated on (address), the expected value at the address (expectedValue), the value to replace the expected value if the MCAS operation succeeds (newValue), and a pointer used to hold a reference to an `MCasHelper` object that was placed at this address (mch). Each value is constant except for mch, which can only transition from `null` to a `non-null` value, after which the it is constant.

```
struct {address, expectedValue,
  newValue, MCasHelper *mch }CasRow
```

- `MCasDescriptor`: a block of memory used to describe an MCAS operation, it is composed of an arbitrary number of `CasRows` followed by the constant 0x1.

```
struct { CasRow[], 0x1 }MCasDescriptor
```

- `MCasHelper` is an object used to hold the value at an address constant until the MCAS operation referenced by it is completed. It contains a single word, `cr`, that holds a reference to a `CasRow` in an `MCasDescriptor`. This `CasRow` is its associated `CasRow` only if the mch word in it holds a reference to that `MCasHelper`.

```
struct {CasRow *cr }MCasHelper
```

- `nThreads` is a global constant representing the number of threads executing in the system.
- `maxFail` is a global constant representing the maximum number of times a thread will retry an operation before making an announcement.
- `pendingOpTable` a global array of length `nThreads` where each thread has a specific position to write an announcement.
- `threadID` is a thread-local value used to identify the position in the `pendingOpTable` that the thread writes a global announcements into.
- `checkID` is a thread-local value used to identify the position in the `pendingOpTable` that the thread checks for a global announcement. Before each check, this value is incremented by one.

IV. Algorithms

This section describes in detail the two phases of execution of our MCAS design. The first phase consists of placing `MCasHelper` objects at each address, if the value at the address matches the expected value. The first phase is complete when the `MCasHelper` pointer of the last `CasRow` holds a non-`null` reference. The second phase consists of replacing each `MCasHelper` with its logical value. For brevity, the bit masking operations are omitted. If a value

[4]Load-link, Validate, Store Conditional; used to ensure the value at an address has not been unknowingly modified.

[5] See Sec. V-C for more details.

read holds an `MCasHelper` bitmark, then the next step would be to unbitmark the local copy before dereferencing it.

Figure 1, presents visual representation of a successful MCAS operation and the relation between a `MCasHelper` and the `CasRow` it references.

A. Algorithm 1 - Begin MCAS operation

This function takes an `MCasDescriptor` object and the address of the last `CasRow` in the object and returns whether or not the MCAS operation was successful. Before a thread commences its own operation, it calls `helpIfNeeded` (Alg. 4) which examines one other thread to determine if that thread is being repeatedly preempted and, if necessary, helps complete that thread's operation.

This function then calls `placeMCasHelper` (L.6) until either all addresses have been acquired, or it has failed to acquire an address. It fails to acquire an address when the current value is not equal to the expected value.

Unlike other approaches, this design does not use a state variable. Rather, the `MCasHelper` pointer of the last `CasRow` determines whether or not the operation is in progress, successful, or failed. If it holds `null`, then the operation is in progress if it holds ~0x0, then the operation has failed otherwise, it is successful. This optimization reduces the number of CAS operations needed by one.

After the result of the operation has been determined, it calls `removeMCasHelper` (L.13), which iterates over each `CasRow` and replaces the `MCasHelper` at each address with its logical value.

Algorithm 1 invokeMCAS $CasRow * mcasp, CasRow * lastRow$

```
 1: helpIfNeeded()
 2: ___thread tl_mcas=mcasp
 3: placeMCasHelper(tl_mcas++, lastRow, true, 0)
 4: repeat
 5:     if lastRow->mch == 0x0 then
 6:         placeMCasHelper(tl_mcas++, lastRow, false, 0)
 7:     else
 8:         break
 9:     end if
10: until tl_mcas == lastRow
11: pendingOpTable[threadID]=null
12: res= (lastRow->mch != ~0x0)
13: removeMCasHelper(res,mcasp, lastRow)
14: safeFree(mcasp)
        return res
```

B. Algorithm 2 - Acquire an address.

This function tries to acquire an address for an MCAS operation by attempting to place a reference to an `MCasHelper`, if the `expectedValue` for the address matches the logical value currently at the address.

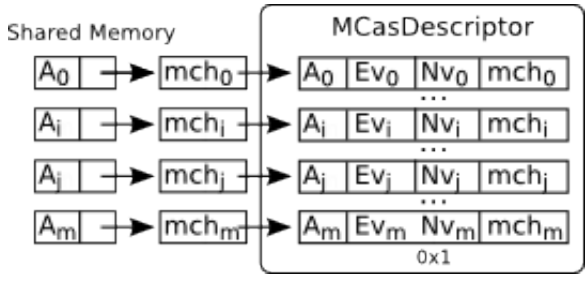

Fig. 1: Example of a Successful MCAS operation

Algorithm 2 void placeMCasHelper $CasRow * cr, CasRow * lastRow, bool firstTime, int rDepth$

```
 1: address=cr->address
 2: eValue=cr->expectedValue
 3: mch= allocateMCasHelper(cr)
 4: cValue= *address
 5: tries=0
 6: while true do
 7:     if tries++ == maxFail  then
 8:         if firstTime then
 9:             cr->mch = null
10:             firstTime = false
11:         end if
12:         pendingOpTable[threadID]= tl_mcas
13:         if rDepth > 0 then
14:             recursive return
15:         end if
16:     end if
17:     if !isMCasHelper(cValue) then
18:         if firstTime then
19:             cr->mch=mch
20:         end if
21:         cValue = CAS(address, eValue, mch)
22:         if cValue == eValue then
23:             if !firstTime then
24:                 cValue = CAS(&cr->mch, null, mch)
25:                 if cValue != null && cValue != mch then CAS(address,
    mch, eValue)
26:                     safeFree(mch)
27:                 end if
28:             end ifreturn
29:         else
30:             continue
31:         end if
32:     else
33:         if cr==cValue->cr then
34:             free(mch)
35:             cValue2 = CAS(&cr->mch, null, cValue)
36:             if cValue2 != null && cValue2 != cValue then
    CAS(address, cValue, eValue)
37:             end ifreturn
38:         else if shouldReplace(eValue, cValue, rDepth then
39:             if firstTime then
40:                 cr->mch=mch
41:             end if
42:             cValue2 = CAS(address, cValue, mch)
43:             if cValue2 == cValue then
44:                 if !firstTime then
45:                     cValue = CAS(&cr->mch, null, mch)
46:                     if cValue != null && cValue != mch then
    CAS(address, mch, eValue)
47:                         safeFree(mch)
48:                     end if
49:                 end if
50:                 break
51:             else
52:                 continue
53:             end if
54:         end if
55:     end if
56:     res=CAS(&cr->mch, null, ~0x0)
57:     if res==null then
58:         CAS(&lastRow->mch, null, ~0x0)
59:     end if
60:     break
61: end while
```

- If the logical value of the address is not equal to the `expectedValue`, then the thread will attempt to set the `MCasHelper` pointers of *cr* and *lastRow* to failed (~0x0) before returning(L. 58).
- If the address holds a reference to an `MCasHelper` object that references *cr* (L. 33), the thread will attempt to set the `MCasHelper` pointer of *cr* to that `MCasHelper` before returning(L. 35).
- Otherwise, if the logical value at the address is equal to the `expectedValue` (L. 21, L. 38), the thread attempts to replace the value with an `MCasHelper`, *mch*, that references *cr* (L. 21, L. 42).

If the thread failed to place *mch* it will use the returned result of the CAS operation to re-evaluate the current value at the address.

If a thread successfully places *mch* (L. 21, L. 42), it will then attempt to associate *cr* with *mch*(L. 24, L. 35). If *cr* is already associated with an `MCasHelper`, this indicates that some other thread completed this MCAS operation, and that *mch* should be removed (L. 25, L. 36, L. 46).

For example in Fig. 1, the `MCasHelper` at address A_j does not match the value of the *mch* pointer in the `CasRow` it references; this situation is examined in detail in Section V-C.

If the number of times a thread has retried its operation is equal to `maxFail`, then it will write its own MCAS operation into a global array(L. 12). Other executing threads are guaranteed to eventually see this operation, and attempt to help complete it. Our association between a `CasRow` and an `MCasHelper` ensures that an operation or portion of an operation will not be repeated when multiple threads attempt to complete it.

In the event a thread is delayed while helping a conflicting MCAS operation, then the thread will return to its own operation. This addresses the scenario where the dependecy between the current MCAS operation and the thread's own operation no longer exists.

We optimize the first time a thread calls `placeMCasHelper` for an MCAS operation (L. 19) to allow an `MCasHelper` to be associated with a `CasRow` before placing the `MCasHelper` at an address. This optimization is available only for the first `CasRow` when the `MCasDescriptor` is not visible to other threads. This further reduces the number of CAS operations needed by one.

C. Algorithm 3 - Should Replace MCasHelper

This algorithm determines whether the logical value of *mch* matches *ev*. First, it checks if either the `expectedValue` or `newValue` of the `CasRow` referenced by *mch* matches *ev* (L.2), if not it returns false. A thread examining mch_0 from Fig. 1 would compare *ev* to Ev_0 and Nv_0.

Next, it calls `helpComplete` to ensure that the referenced MCAS operation is no longer in progress and to determine the result of the operation.

If the MCAS operation was successful and *mch* is associated with its CasRow, then `newValue` is the logical value of *mch*. Otherwise, `expectedValue` is the logical value of *mch*. For example in Fig. 1, the logical value of mch_i would be Nv_i and the logical value of mch_z would be Ev_j.

A boolean is returned indicating if the logical value matches *ev* (L.6,L.10).

D. Algorithm 4 - Help delayed thread

Before a thread attempts an operation, it checks one other thread to see if that thread needs help completing its operation. This check is performed by examining the `checkId` position of the `pendingOpTable` (L. 2). Before each check the thread will increment `checkId` by one. This ensures that all positions in the table will be examined after `numThread` calls to this function. This scheme is derived from the helping approach presented by Kogan et al. in [11]. If a delayed operation is found, then the thread invokes `helpComplete` before returning.

Algorithm 3 shouldReplace $void * ev, MCasHelper * mch, intrDepth$

```
1: cr = mch->cr
2: if cr->expectedValue != ev && cr->newValue != ev then return
   false
3: else
4:     res=helpComplete(cr, rDepth+1)
5:     if res && (cr->mch == mch) then
6:         if (cr->newValue == ev) then return true
7:         elsereturn false
8:         end if
9:     else
10:        if cr->expectedValue == ev then return true
11:        elsereturn false
12:        end if
13:    end if
14: end if
```

Algorithm 4 helpIfNeeded *intrDepth*

```
1: checkId=(checkId+1)%nThreads
2: cr=pendingOpTable[checkId]
3: if cr != null then
4:     helpComplete(cr, 0)
5: end if
```

E. Algorithm 5 - Help another thread

This function allows a thread to help complete another thread's delayed MCAS operation.

In the event the recursive calls to `helpComplete` is greater then the number of threads, then there must be no dependency between this threads operation and the operation it is currently helping. The thread will then return back to its own operation, increment its failCount, and attempt to acquire the address again.

The thread will first search for the last `CasRow`, *lastRow*, of the `MCasDescriptor`, allowing it determine if the operation has been completed or if it is still in progress. This is done by iterating through each `CasRow` until an end marker is reached, in our implementation it is a `CasRow` with an address of 0x1. The thread repeatedly calls `placeMCasHelper` until the operation is complete, indicated by the *lastRow* holding a non-null value. Finally it returns whether the operation was successful or if it failed.

Algorithm 5 helpComplete $CasRow * mcas, intrDepth$

```
1: if rDepth > nThreads then
2:     recursive return
3: end if
4: lastRow=cr
5: while lastRow->address != 0x1 do
6:     lastRow++
7: end while
8: lastRow−
9: repeat
10:    if  lastRow->mch == 0x0 then
11:        placeMCasHelper(mcas++, lastRow, false, rDepth)
12:        if mcas->mch == ~0x0 then
13:            break
14:        end if
15:    else
16:        break
17:    end if
18: until mcas == lastRow
       return (lastRow->mch != ~0x0)
```

F. Algorithm 6- Remove MCasHelpers

This function removes the `MCasHelper` descriptors that were placed during this MCAS operation. For each `CasRow`

in the MCasDescriptor, it attempts to replace the associated MCasHelper with its logical value. If the MCAS operation was successful, then each MCasHelper is replaced by the newValue from its CasRow. Otherwise, it is replaced by the expectedValue.

V. Correctness

A. Semantics

An MCAS operation is successful and subsequently replaces the value at each address with a new value, if each address matches the respective expected value. To provide correctness, this must appear to happen atomically, such that overlapping operations cannot read a new value at one address and then read an old value at another address. To guarantee this behavior we use *linearizability* as our main correctness guarantee. Linearizability is a correctness property that requires for each operation call to "appear to take effect instantaneously at some moment between its invocation and response" [9, p. 54]. If a class is composed of linearizable functions, then a legal sequential history of executions can be derived from every concurrent execution. In the derived sequential history, operations are ordered according to the moment of time of their invocation and response. Operations with overlapping invocation and response events, are ordered according to their linearization points. We show that concurrently executing MCAS operations are linearizable.

Another property our algorithm provides is wait-freedom, which is a progress condition that guarantees that each operation completes in a finite number of steps. This differs from lock-freedom, which guarantees that at least one operation completes. Providing wait-free execution is important for systems where concurrency and real-time response are critical. We show that our algorithm is wait-free by determining the maximum number of steps it takes for a function call to return. This upper bound is derived from the known total number of threads and can be fine-tuned by a user-defined threshold value.

Below we present a set of lemmas in support of our hypothesis that our MCAS algorithm is linearizable, ABA-free, and wait-free. We argue that in no case does our design deviate from its intended behavior, each step of the MCAS operation completes in a finite number of steps, and that cases of ABA are avoided.

B. Linearizability

This section introduces a set of lemmas and theorems that show our design is linearizable.

Lemma 1. *After initialization an* MCasDescriptor *object remains constant, except for the* MCasHelper *pointer in each* CasRow*.*

Lemma 2. *Once an* MCasHelper *object is placed at an address, its internal pointer is constant.*

Lemma 3. *The* MCasHelper *pointer word of a* CasRow *can only transition from* null *to a* non-null *value.*

Lemma 4. *The first* CasRow *of an* MCasDescriptor *has its* MCasHelper *pointer set before any other* CasRow *in the* MCasDescriptor *and for $i > 1$, if the i^{th}* CasRow *has its* MCasHelper *pointer set, then the $i^{th} - 1$* CasRow *has its* MCasHelper *pointer set.*

These lemmas provide insight into how shared objects are modified, by specifying the fields which are subject to change and the operations performed on those fields. Lemmas 1, 2, and 3 are supported by the semantics of the algorithms presented in Sec. IV.

Each object is initialized[6] while the object is thread-local[7] and modifications made after the object is no longer thread-local are done through a CAS operation. The point when an MCasDescriptor is no longer thread-local is when the first MCasHelper for it has been placed at an address or it is placed into the pendingOpTable. Specifically, lines 12, 21 and 42 from Alg. 2, are the updates used to modify shared objects; the expected values of these CAS operations are constant, such that after the first successful CAS update, any further CAS operations will fail.

Lemma 5. *The* CasRow *referenced by an* MCasHelper *holds the value replaced by the* MCasHelper*,* expectedValue*, and the value to replace the* MCasHelper*,* newValue*, if the MCAS operation has succeeded.*

Lemma 6. *A thread can correctly determine whether to use the* expectedValue *or* newValue *from a* CasRow *for the* logical value *at an address holding an* MCasHelper*.*

Theorem 1. *Our MCAS algorithm is linearizable.*

Lemma 5 is supported by Alg. 2 and Lemmas 1, 2. Together they ensure that the MCasHelper has its reference set before being placed at the address, the reference does not change, and that the expectedValue and newValue word of the CasRow remain constant. Additionally, the CAS used to place the MCasHelper uses the expectedValue word from that CasRow as the expected value parameter, which guarantees that if the MCAS operation failed, the MCasHelper can be replaced by the value that it replaced.

To support Lemma 6, consider Alg. 3, where a thread will identify from an MCasHelper, the CasRow it references and the last CasRow of the operation. The MCAS operation is in progress until the last MCasHelper pointer of the CasRow holds a non-null value.

To support Lemma 4, consider Alg. 1 and Alg. 5, which iteratively call Alg. 2 on each CasRow in the MCasDescriptor, with no method by which a CasRow can be bypassed. If a thread encounters an MCasHelper, *mch*, then to complete the MCAS operation it must iterate from the CasRow referenced by the *mch* to the last CasRow of the MCasDescriptor.

To support Theorem 1, we identify the linearization

[6]This requires a sequential consistent memory model

[7]An object is considered thread-local if only one thread holds a reference to that object.

point of an MCAS operation, which is the CAS operation that sets the `MCasHelper` pointer of the last `CasRow`. If the `MCasHelper` pointer of the last `CasRow` is set then either, a thread has set it to a failed marker (~0x0) or it references an `MCasHelper`.

Lemma 4 shows that if the `MCasHelper` pointer of the last `CasRow` references an `MCasHelper` then each `CasRow` in the `MCasDescriptor` has acquired an address, meeting the criteria for a successful MCAS. Lemmas 5 and 6 support our claim of linearizability by showing that if an address holds an `MCasHelper` then the determined logical value of that object is linearizable.

Any thread that determined a logical value of an `MCasHelper`, can be ordered before or after the MCAS operation based on the value read from the last `CasRow`. Additionally, if two MCAS operations have overlapping addresses, then they are ordered based on which operation acquires the lowest common address first. The other operation will be forced to help complete this operation before it retries to acquire the address.

If a thread is accessing an address that has been acquired by an MCAS operation, then the logical value at the address is determined as follows:

- If the operation is in progress , then expected value is the logical value of the address.
- If the operation successfully completed, then the new value of the `CasRow` is the logical value only if the `CasRow` is associated with that `MCasHelper`.
- Otherwise, the operation failed or the `CasRow` is associated with a different `MCasHelper`[8], and the expected value of the `CasRow` is the logical value.

C. The ABA Problem

This section presents an informal reasoning about the ABA problem and how it applies to MCAS algorithms in general. Here, we argue that if a thread helps to complete another thread's MCAS operation, then this does not introduce undesired behavior as a result of thread delay or the ABA problem. In contrast to the designs presented by Harris et al. and Sundell, our design places references to `MCasHelper` objects at addresses instead of references directly to a `CasRow` or `MCasDescriptor`. In the aforementioned designs, there must be a mechanism to distinguish between a referenced placed during the operation and one placed after the operation has been completed.

Fig. 2 and 3 presents an example of the ABA problem occurring when a thread helps bring another thread's MCAS operation to completion without a mechanism to prevent ABA. Fig. 3 presents the expected history of values and the history of values when ABA occurs on address a_i.

In this example the calling application intended for the value at the address to transition from Ev_i to Nv_i, and back to Ev_i. The execution history of an ABA prone MCAS algorithm, changes the intended behavior of the calling application by introducing a second transition

[8]See Section V-C for details

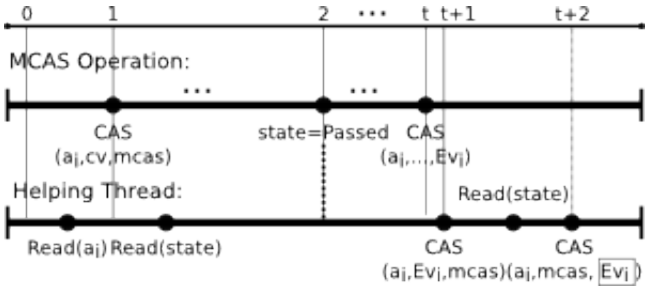

Fig. 2: Example of ABA

T:	0	1	2	...	t	t+1	t+2
Expected:	Ev_i	mcas	Nv_i	...	Ev_i	Ev_i	$\mathbf{Ev}_i$
ABA:	Ev_i	mcas	Nv_i	...	Ev_i	mcas	$\mathbf{Nv}_i$

Fig. 3: History of values at a_i

from Ev_i to Nv_i. This incorrect transition can introduce hard to detect bugs in the application, underscoring the importance of ABA freedom.

The approach presented by Harris et al. describes a "double compare, single swap" algorithm that prevents the ABA problem, but requires explicit memory barriers and additional memory management. Sundell made no claims of ABA freedom, and using the presented example, it can be shown his design is prone to ABA. Our design avoids this issue by placing a references `MCasHelper` objects instead of `MCasDescriptor` objects. To distinguish between an `MCasHelper` placed during the operation and one place after, an association is made between a `CasRow` and an `MCasHelper`.

Theorem 2. *The presented algorithm is ABA free.*

If a thread determines it must help complete another thread's MCAS operation in order to make progress with its own, then it will attempt to acquire the rest of the addresses for that MCAS operation. From Lemma 4, the thread is aware that all previous addresses have been acquired and from Lemma 3, if an address has already been acquired, it cannot be re-acquired after the operation has been completed. If a thread places an `MCasHelper` at an address for a `CasRow` that is already associated with an `MCasHelper`, then, by Lemma 3, it will fail to associate the `CasRow` with its `MCasHelper`. This failure will cause the `MCasHelper` to be replaced by the `expectedValue` word of the `CasRow`.

The ABA problem can also occur if shared objects are incorrectly reused; to prevent this, a memory management scheme, such as hazard points [14] or reference-counting [5] must be used.

In testing, we used a reference-counting scheme to ensure that objects are not referenced by another thread before they are reused. When a thread reads a reference to an object, it will increment that object's reference count and then perform a second read to ensure that the contents of the address read have not changed. If they have, then the reference count is decremented, and the process is repeated until it is successful. In our implementation, if

the value of the address changes, then this causes the `failCount` to be incremented and the new value of the address is examined. This does not effect the wait-free progress guarantee our algorithm provides.

D. Progress Guarantee

This section supports our claim that the presented algorithm is wait-free by describing the maximum number of steps our design takes to complete an MCAS operation. We start by showing our design is lock-free, then examine the case where a thread must retry its operation, and derive an upper bound on the number of steps to complete an operation.

Lemma 7. *Addresses in an MCAS operation are finite and sorted in a descending order.*

Theorem 3. *Our design is lock-free*

To prove our algorithm is lock-free we start with the following observations: From Lemma 7, if each address could be acquired in a finite number of steps, then the MCAS operation completes in a finite number of steps. Addresses are sorted in a descending order, which prevents possibility of cyclical dependency of MCAS operations and places a physical bound on length of recursive helping. Further the depth of recursive helping can be limited to the number of threads, as this implies that the dependency between the threads own operation the ones it is helping have changed.

If a thread successfully acquires an address, then that thread has made progress toward completing its operation. If A is the number of acquired addresses and M the number of addresses in the operation, then there are M - A addresses left to acquire.

If the thread failed to acquire an address, then this implies another thread may have made progress toward completing its operation. If the result of the failed CAS returned another thread's `MCasHelper`, then that other thread made progress. If the value was not an `MCasHelper`, progress is determined by the result of the next CAS operation. Suppose that this subsequent CAS also failed and returned a non-`MCasHelper`, then there must have been some intermediate `MCasHelper` at the address; the thread that placed the intermediate `MCasHelper` made progress.

If an address holds an `MCasHelper` whose MCAS operation is in progress, than from Alg. 5, the thread will be able to complete that operation, allowing it to attempt to acquire the address. Failing to acquire an address, implies that another thread has made progress in its operation and that no value can prevent a thread from attempting to acquire an address, which supports our claim of lock-freedom 3. To show our algorithm is wait-free, we examine the case where a thread fails to acquire an address. If the result of a failed CAS matches the expected value, then the thread must retry. Otherwise, the MCAS operation can be allowed to return false.

Lemma 8. *The helping scheme ensures a thread cannot be indefinitely being prevented from acquiring an address.*

Lemma 9. *The maximum number of attempts to acquire an address is equal to* $maxFail + nThreads^2$.

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

To limit the number of times a thread must re-attempt to acquire an address, we use an announcement scheme [8], to indicate that a thread has been delayed. If the number of attempts to acquire and address is equal to `maxFail`, a user defined constant, then then thread will make an announcement indicating that it is delayed. From Alg. 4, before beginning an MCAS operation, each thread checks another thread for such an announcement, helping complete an operation if necessary. In the worst case, each other thread will have just checked the delayed thread and found that it did not need help. Allowing each of them to complete `nThreads` more MCAS operations before checking that thread again, Lemma 9.

Together, Lemma 9 and Theorem 3 shows that a thread can acquire all addresses in an MCAS operation, thus completing it, in a finite number of steps by ensuring that if a thread is continually prevented from acquiring an address, then the threads that are preventing it will help that thread complete its operation when they observe a delayed thread, Theorem 4.

VI. Performance Evaluation

In this section, we evaluate the scalability and performance of our algorithm and compare it with the the lock-free MCAS (LFMCAS) presented by Harris et al. Unfortunately, when tested, Sundell's wait-free MCAS exhibited behavior that produced inconsistencies in the testing methodology, invalidating the test results. All implementations were provided by their respective authors [20], [7]. In our experimental evaluation, we employ a micro-benchmark to simulate the effect that high contention has on scalability and performance. This micro-benchmark consisted of a tight loop in which the value of each word in a multi-word object is read, then using MCAS atomically incremented by a constant. To explore the case where there is a high degree of parallelism but a low degree of contention in the system, we implemented an MCAS-based sorted double-linked list. The design uses a four word long MCAS operation to atomically insert or remove values. The performance evaluation of both benchmarks are presented below. All tests are conducted on a 64-core ThinkMate RAX QS5-4410 server running Ubuntu 12.04 LTS. It is a NUMA system with four AMD Opteron 6272 CPUs (16 cores per chip @2.1 GHz) and 314 GB of shared memory. All executables were compiled with GCC 4.7 (with the option -std=c++0x to enable C++ 11 support).

A. Testing Methodology

For each benchmark and MCAS algorithm tested, a separate executable file was generated. In each benchmark, a main thread initialized all global values and created a

set of worker threads. When each worker thread is ready, the main thread signals them to begin execution. After sleeping for a specified amount of time, the main thread signals the end of execution. The sum of the total number of operations completed by each thread was logged and the average of fifteen runs is used in the following graphs.

B. Benchmark: Multi-word object

In the multi-word object benchmark each thread repeatedly tries to increment the value of each word in the object by 16[9]. This is accomplished by performing an MCAS read operation [10] on each word of the object, then calling MCAS to replace the values read with the new values calculated. Our performance metric for this test was the total number of MCAS operations completed by all threads during execution. Our evaluation includes measuring the effect that the number of executing threads and the size of of the multi-word object has on performance.

Fig. 4 presents a set of representative graphs based on our first benchmark. Graphs 4a, 4b, and 4c depict the effects of increasing the number of threads updating a shared multi-word object. The performance results show that, in this scenario, on average the WFMCAS performs 10% more operations per second when compared to the LFMCAS. When the number of threads is 16, on average the WFMCAS performs 35.4% more operations per second. Increasing the number of threads to 32 and 64, we perform 50.3% and 77.1%, respectively, more operations than the LFMCAS.

Not only does our design achieve a higher throughput of operations than the LFMCAS, but it also provides a stronger guarantee of progress, wait-freedom. We attribute the difference in performance to how we manage the ABA problem; where the LFMCAS uses auxiliary data structures and memory barriers, our design uses an association. This association allows us to reduce the number of CAS operations required by our algorithm to 3M-1, while the LFMCAS requires 3M+1 CAS operations in addition to depending on memory barriers for correctness.

Graph 4d presents the effect on performance of increasing the number of words in the object while keeping the number of executing threads at 64. In this graph, WFMCAS performs on average 67.8% more operations than the LFMCAS, indicating our design scales better than the LFMCAS, as the size of the MCAS operation increases.

For MCAS operations on a large number of addresses, the LFMCAS helping scheme requires a thread to load each address in the operation to determine if it holds a reference to the operation. Depending on where these addresses are located, this may generate a large number of cache misses. Our WFMCAS design's association between `CasRow` and `MCASHelper` objects enables a thread to iterate through the MCAS operation instead of loading each address, to determine if an address has been acquired or not.

These graphs revealed that as the number of threads grows, the number of completed MCAS operations performed by both algorithms decreases. This decrease in performance is a result of the CAS operation that these algorithms are built on. Unlike atomic reads and writes, the CAS operation provides an infinite consensus number [8]. This allows effective interprocess coordination and enables a developer to correctly reason about the value of an address before and after the operation is applied.

The CAS operation is often implemented in hardware using techniques that lock the memory bus, creating a bottleneck when threads access or modify memory.

The following benchmark explores the case where the contention on each memory word is lower and the amount of work done outside of the MCAS algorithm is higher than in the previous benchmark.

C. Sorted-Double Linked List

In the sorted double-linked list (SDLL) benchmark, each thread repeatedly tries to insert and delete elements from the data structure. The probability of a thread performing an insert operation was varied between 25% and 100%. Each thread randomly generates two integers; the first is used to select whether to perform an insert or delete operation, and the second is used as the operand of the selected operation.

To perform an insert or delete operation, a thread will linearly search the queue for a value that is greater than or equal to the specified value. Then using a four-word-long MCAS operation attempt to apply its operation. For example, when inserting *node* between *parent* and *child*, an MCAS operation will be invoked to change the *parent->next* to *node*, *child->previous* to *node*, *node->previous* to *parent*, and *node->previous* to *child*.

This design uses a four word long MCAS operation along with the announcement scheme described earlier to provide a wait-free progress guarantee, if the underlying MCAS operation is also wait-free.

In contrast to the first benchmark, which performed minimal work between calls to the MCAS operation, this benchmark generates two random values and then performs a linear search on the data structure. This reduces the impact that the MCAS algorithm has on performance. Additionally, this benchmark distributes the synchronization across various regions of memory, as opposed to the first benchmark where synchronization is performed on a single region of memory.

Our experiments revealed that varying the ratio of insert to remove operations had minimal effect on the overall scalability of the SDLL; graphs 5a and 5b are representative of our experiments. The graphs show that as the number of threads increases, both implementations scale equally well and on average, over all tests, the WFMCAS performs 2% more operations per second than

[9]Incrementing by 16 ensures that the two least significant bits are always 0.

[10]An MCAS read function is designed to return the logical value of a descriptor object that may be at an address.

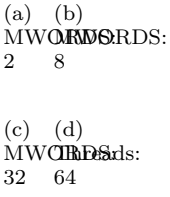

Fig. 4: Multi-word Test Results (log scale)

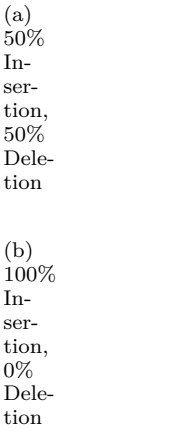

Fig. 5: Sorted-Double Linked List

the LFMCAS. An explanation for the lack of significant speedup in SDLL, can be found by examining the run time of the application with respect to Amdahl's law [1]. The cost of performing the MCAS operation is eclipsed by both the random number generation and the O(n) search performed on the list. When the list is even moderately long, 84% of the execution time is spent searching it. These benchmarks revealed that when implemented in a practical data structure, not only does our design allow the data structure designer to use MCAS a wait-free approach, but they can do so without having to sacrifice performance.

VII. Acknowledgements

Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04- 94AL85000.

VIII. Conclusion and Future Work

In conclusion, we presented the first multi-word compare-and-swap algorithm that is ABA-free and wait-free in all scenarios of execution. It provides 77.1% improved performance in scenarios of high contention and comparable performance to other approaches all other scenarios of execution. We presented an MCAS based sorted double linked list algorithm, that behaves in a wait-free manner with the presented MCAS algorithm, and degrades to lock-free when using other approaches. In contrast to other practical designs which requires two bits to be reserved from each word, our design requires a single bit, making it practical for a wider range of applications. We supported these claims of wait-freedom, ABA-freedom, and improved performance, with an informal proof and a series of benchmark tests.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, 1967. ACM.

[2] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.*, 15(2):134–165, May 1997.

[3] H. Attiya and E. Hillel. Highly concurrent multi-word synchronization. *Theor. Comput. Sci.*, 412(12-14):1243–1262, Mar. 2011.

[4] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, SPAA '93, pages 261–270, New York, NY, USA, 1993. ACM.

[5] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr. Lock-free reference counting. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, PODC '01, pages 190–199, New York, NY, USA, 2001. ACM.

[6] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

[7] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 265–279, London, UK, UK, 2002. Springer-Verlag.

[8] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, Nov. 1993.

[9] M. Herlihy. *The art of multiprocessor programming.* Elsevier/-Morgan Kaufmann, Amsterdam London, 2008.

[10] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, PODC '94, pages 151–160, New York, NY. ACM.

[11] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. *SIGPLAN Not.*, 47(8):141–150, Feb. 2012.

[12] Y. Liu and M. Spear. A lock-free, array-based priority queue. *SIGPLAN Not.*, 47(8):323–324, Feb. 2012.

[13] F. Meawad, M. Schoeberl, K. Iyer, and J. Vitek. Real-time wait-free queues using micro-transactions. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 1–10, New York, NY, USA, 2011. ACM.

[14] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

[15] M. Moir. Transparent support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, WDAG '97, pages 305–319, London, UK, UK, 1997. Springer-Verlag.

[16] C. Purcell and T. Harris. Non-blocking hashtables with open addressing. In *Proceedings of the 19th international conference on Distributed Computing*, DISC'05, pages 108–121, Berlin, Heidelberg, 2005. Springer-Verlag.

[17] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM.

[18] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *Proceedings of the 9th international conference on High performance computing for computational science*, VECPAR'10, pages 1–25, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] D. D. Steven Feldman, Pierre LaBorde. Concurrent multi-level arrays: Wait-free extensible hash maps. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 155 – 163, July 2013.

[20] H. Sundell. Wait-free multi-word compare-and-swap using greedy helping and grabbing. *International Journal of Parallel Programming*, 39:694–716, 2011. 10.1007/s10766-011-0167-4.

[21] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-free linked-lists. *SIGPLAN Not.*, 47(8):309–310, Feb. 2012.