

# 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  
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 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). Our performance evaluation reveals that on average our wait-free MCAS algorithm performs 8.3% faster than other practical approaches in all tested scenarios.

## Categories and Subject Descriptors

D.4.1 [Process Management]: Concurrency; Mutual Exclusion; Synchronization

## Keywords

Wait-Free, Lock-Free, Non-Blocking, Concurrent, MCAS, CAS

## 1. INTRODUCTION

On-chip parallelism is expected to be the primary area of parallelism growth in future multiprocessor systems [15].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

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<sup>1</sup>).

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. [6]. 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 [16], linked lists [18], queues [11], and et al. These data structures often depend on atomic primitives, such as atomic read, atomic store, and Compare-and-Swap (CAS)<sup>2</sup> 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 ne-

<sup>1</sup>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.

<sup>2</sup>An operation with infinite consensus number in the wait-free/lock-free hierarchy

cessity. A wait-free, ABA-free MCAS operations allows a developer to express the semantics of these advanced algorithms without the underlying 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 [5]. 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 [13], 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, leading to a high cache miss rate.
- A lock-free, array-based priority queue implementation [10], 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 [5], 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 [14].

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<sup>3</sup> 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.

<sup>3</sup>An object that allows an interrupting thread to help an interrupted thread to complete successfully [3].

- 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 than 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.

## 1.1 Road Map

The remainder of this paper is structured as follows: Section 2 describes other MCAS implementations. Section 3 and 4 provide detailed descriptions of how the algorithm is implemented. Section 5 provides an informal proof that our model behaves properly in all cases and that of our approach meets all claims made. In Section 6 we present experimental data that show how different implementations compare in different use case scenarios. We conclude in Section 7.

## 2. RELATED WORK

Israeli et al. present a lock-free and disjoint-access parallel MCAS algorithm [8]. 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 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 before proceeding with its own operation. This algorithm is dependent on the LL/VL/SC primitive<sup>4</sup>, 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 [1]. In contrast to [8], 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. Like [8] this design requires the LL/VL/SC primitive. A simplified lock-free version of this algorithm was presented by Moir [12]. Attiya et al. [2] have also presented improvements upon this design.

Harris et al. [4] 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 [8] and [1] their design shows a significant increase in performance and portability.

<sup>4</sup>Load-link, Validate, Store Conditional; used to ensure the value at an address has not been unknowingly modified.

Sundell [17] proposes a wait-free MCAS algorithm based on a greedy helping scheme. 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 [4], Sundel 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<sup>5</sup>. Additionally, the algorithm degrades to lock-freedom in the case where the value at an address continually changes between values indicating the thread should retry.

### 3. STRUCTURES

This section describes the global variable, 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.

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

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

- **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 (`MCasHelperPointer`).

These four words are used to describe a particular step of the MCAS operation. All except for the **MCasHelperPointer** are constants. The **MCasHelperPointer** can only go from `null` to a `non-null` value, as such it will not change more than once.

```
CasRow { address, expectedValue,
         newValue, MCasHelperPointer };
```

- **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 **MCasHelperPointer** 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.

<sup>5</sup> See Sec. 5.3 for more detail.

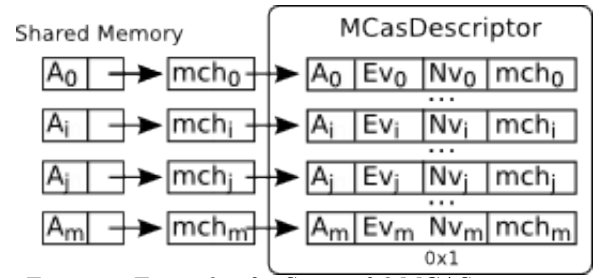


Figure 1: Example of a Successful MCAS operation

- **threadID** is a thread local value used to identify the position in the `pendingOpTable` that the thread writes 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.

### 4. 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 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.

#### 4.1 Algorithm 1 - Begin MCAS operation

This function takes an **MCasDescriptor** object and the address of the last **CasRow** in the object and its return value indicates 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. After `nThreads` calls to `helpIfNeeded` all threads will have been examined this thread.

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.

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.

#### 4.2 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**,

---

**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)
   return res

```

---

if the expected value (*ev*) for the address matches the logical value currently at the address (*cv*).

- If the logical value of the address is not equal to the expected value, then the thread will attempt to set the `MCasHelper` pointers of `cr` and `lastRow` to failed (`~0x0`) before returning(L. 56).
- If the address holds a reference to an `MCasHelper` object that references `cr` (L. 31), the thread will attempt to set the `MCasHelper` pointers of `cr` to that `MCasHelper` before returning(L. 33).
- Otherwise, if the logical value at the address is equal to the expected value (L. 18, L. 38), the thread attempts to replace the value with an `MCasHelper`, `mch`, that references `cr` (L. 18, 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. 18, L. 42), it will then attempt to associate `cr` with `mch`(L. 21, L. 33). If the the thread fails to associate them, this indicates that some other thread completed this MCAS operation, and that `mch` should be removed (L. 23, L. 35, L. 47).

For example in Fig. 1, the `MCasHelper` at address  $A_j$  does not match the value of the `MCasHelperPointer` in the `CasRow` it references, the cause of this is examined in section 5.3.

We optimize the first time a thread calls `placeMCasHelper` for an MCAS operation (L. 16) 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 reduces the number of CAS operations needed by one.

If a thread is unable to complete its operation in `maxFail` number of attempts, 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 will not be repeated when multiple threads attempt to complete it.

### 4.3 Algorithm 3 - Should Replace MCasHelper

This algorithm determines whether the logical value of `mch` matches `ev`. First, it checks if either the expected value or new value 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 MCAS operation is no longer in progress and to get the result of the operation. If the MCAS operation was successful and `mch`

---

**Algorithm 2** `void placeMCasHelper CasRow * cr, CasRow * lastRow, bool firstTime, intrDepth`

---

```

1: address=cr->address
2: ev=cr->expectedValue
3: mch= allocateMCasHelper(cr)
4: cv= *address
5: tries=0
6: while true do
7:   if tries++ == maxFail && pendingOpTable[threadID]==null then
8:     if firstTime then
9:       cr->mch = null
10:      firstTime = false
11:    end if
12:    pendingOpTable[threadID]= tl_mcas
13:  end if
14:  if !isMCasHelper(cv) then
15:    if firstTime then
16:      cr->mch=mch
17:    end if
18:    cv = CAS(address, ev, mch)
19:    if cv == ev then
20:      if !firstTime then
21:        cv = CAS(&cr->mch, null, mch)
22:        if cv != null && currentValue != mch then
23:          cv = CAS(address, mch, ev)
24:        end if
25:      end if
26:      break
27:    else
28:      continue
29:    end if
30:  else
31:    if cr==mch->cr then
32:      mch=cv
33:      cv = CAS(&cr->mch, null, mch)
34:      if cv != null && cv != mch then
35:        cv = CAS(address, mch, ev)
36:      end if
37:      break
38:    else if shouldReplace(ev, cv, rDepth then
39:      if firstTime then
40:        cr->mch=mch
41:      end if
42:      cv2 = CAS(address, cv, mch)
43:      if cv2 == currentValue then
44:        if !firstTime then
45:          cv = CAS(&cr->mch, null, mch)
46:          if cv != null && cv != mch then
47:            cv = CAS(address, mch, ev)
48:          end if
49:        end if
50:        break
51:      else
52:        continue
53:      end if
54:    end if
55:  end if
56:  CAS(&lastRow->mch, null, ~0x0)
57:  CAS(&cr->mch, null, ~0x0)
58:  break
59: end while

```

---

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).

### 4.4 Algorithm 3 - Remove MCasHelpers

This function removes the 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`

---

**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

```

---

Otherwise, it is replaced by the `expectedValue`

## 4.5 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 [9]. If a delayed operation is found, then the thread invokes `helpComplete` before returning.

---

**Algorithm 4** helpIfNeeded *intrDepth*

---

```

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

```

---

## 4.6 Algorithm 5 - Help another thread

This function allows a thread to help complete another thread's delayed MCAS operation. 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. Then the thread repeatedly calls `placeMCasHelper` until the operation is complete

---

**Algorithm 5** helpComplete *CasRow \* mcas, intrDepth*

---

```

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

```

---

# 5. CORRECTNESS

## 5.1 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” [7, 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.

## 5.2 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  $i^{\text{th}}$  CasRow has its MCasHelper pointer set, then the  $i^{\text{th}} - 1$  CasRow has its MCasHelper pointer set.*

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.*

To support Theorem 1, we identify the linearization 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`<sup>6</sup>, and the expected value of the `CasRow` is the logical value.

### 5.3 The ABA Problem

This section presents the ABA problem and how it is handled in our MCAS algorithm. Here we argue that if a thread helps to complete another thread's MCAS operation, then this does not introduce undesired behavior.

Our design places references to `MCasHelper` objects at addresses instead of references directly to a `CasRow` or `MCasDescriptor` because that such designs are prone to the ABA problem. Harris et al. avoided the ABA problem by using a "double comp rare, single swap" algorithm at the cost of depending on explicit memory barriers and additional memory management. Sundel makes no claim of ABA freedom, and it can be shown that his presented design is ABA prone. Fig. 2 presents an example of the ABA problem occurring when a thread helps bring another thread's MCAS operation to completion. Fig. 3 presents the expected history of the value at address  $a_i$  and history of the values at the address when ABA occurs.

In designs that place references directly to the operation, there are no mechanism to distinguish between a reference to a descriptor placed during the operation and a reference placed after the operation has been completed. Our design avoids this by placing a reference to a descriptor object, `MCasHelper`, instead of the MCAS descriptor object, `MCasDescriptor`. To distinguish between an `MCasHelper` placed during the operation and one place after, an association is made between a `CasRow` and an `MCasHelper`.

<sup>6</sup>See Section 5.3 for details

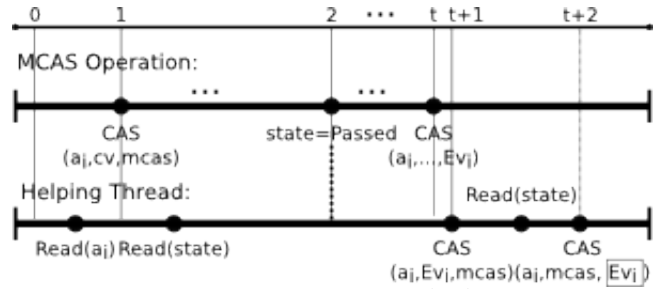


Figure 2: Example of ABA

T:	0	1	2	...	t	t+1	t+2
Expected:	$Ev_i$	mcas	$Nv_i$	...	$Ev_i$	$Ev_i$	<b><math>Ev_i</math></b>
ABA:	$Ev_i$	mcas	$Nv_i$	...	$Ev_i$	mcas	<b><math>Nv_i</math></b>

Figure 3: Example of ABA

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`.

### 5.4 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 when 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. If a thread successfully acquires an address, then that thread has made progress toward completing its operation. Failing to acquire an address, implies some other thread has acquired or released the address. Both cases supporting our claim of lock-freedom, Theorem 3, since one thread is always making progress.

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  $\text{maxFail} + n\text{Threads}^2$ .*

If a thread repeatedly fails to acquire an address, then it 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  $n\text{Threads}$  more MCAS operations before checking that thread again, Lemma 9.

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

Our algorithm is wait-free because 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.

## 6. EVALUATION

In this section we evaluate our algorithm’s performance in two benchmarks.

### 6.1 Test Environment

All tests were run on a SuperMicro server, with four sixteen-core AMD Opteron 6272 processors at 2.1 GHz, and a total of 314 gigabytes of RAM. The machine was running 64-bit Ubuntu Linux version 11.04, and all code was compiled with `g++4.7`, with level three optimizations enabled.

In the presented benchmark we compare the performance of our wait-free algorithm against the lock-free MCAS presented by Harris et al. When tested, Sundell’s wait-free MCAS exhibited behavior that produced inconsistencies in the testing methodology, invalidating the test results. Both implementations were provided by their respective authors [17, 5].

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.

### 6.2 Multi-word object

The first benchmark consisted of each thread attempting to update a shared multi-word object. In this benchmark, we examined the effect of increasing the number of threads and the length of the multi-word object.

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.

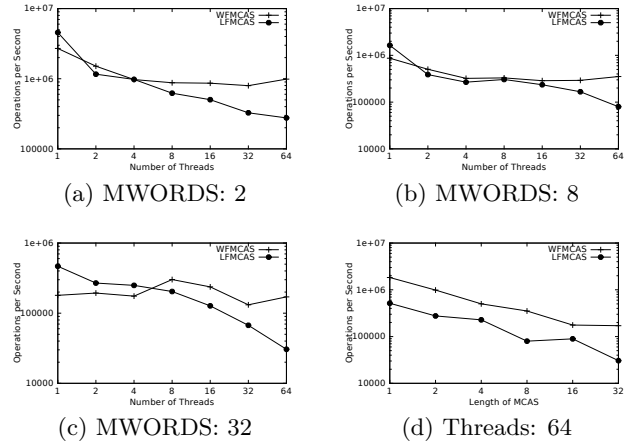


Figure 4: Multi-word object

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.

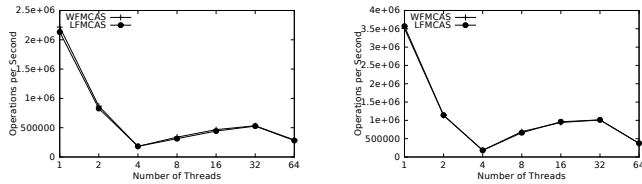
### 6.3 Sorted-Double Linked list

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->next` 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 op-

eration is also wait-free.



(a) 50% Insert, 50% Remove (b) 100% Insert, 0% Remove  
Figure 5: Sorted-Double Linked list

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 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. 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.

## 7. CONCLUSION AND FUTURE WORK

This paper presented a wait-free, ABA-free MCAS algorithm that is practical for a variety of applications. Our implementation provides the progress guarantee of wait-freedom while providing improved performance over a lock-free MCAS implementation. We discussed the relevance of this work and its applicability in the real-world. Future work involves generalizing our design to provide additional functionality and flexibility by allowing the developer to specify more complex operations on each address.

## 8. REFERENCES

- [1] 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.
- [2] H. Attiya and E. Hillel. Highly concurrent multi-word synchronization. *Theor. Comput. Sci.*, 412(12-14):1243–1262, Mar. 2011.
- [3] 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.
- [4] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.
- [5] 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.
- [6] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, Nov. 1993.
- [7] M. Herlihy. *The art of multiprocessor programming*. Elsevier/Morgan Kaufmann, Amsterdam London, 2008.
- [8] 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, USA, 1994. ACM.
- [9] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. *SIGPLAN Not.*, 47(8):141–150, Feb. 2012.
- [10] Y. Liu and M. Spear. A lock-free, array-based priority queue. *SIGPLAN Not.*, 47(8):323–324, Feb. 2012.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-free linked-lists. *SIGPLAN Not.*, 47(8):309–310, Feb. 2012.