

# Wait-Free Multi-Word Compare-And-Swap using Greedy Helping and Grabbing

H. Sundell<sup>1</sup>

<sup>1</sup> School of Business and Informatics, University of Borås, Borås, Sweden

**Abstract**—We present a new algorithm for implementing a multi-word compare-and-swap functionality supporting the *Read* and *CASN* operations. The algorithm is wait-free under reasonable assumptions on execution history and benefits from new techniques to resolve conflicts between operations by using greedy helping and grabbing. Although the deterministic scheme for enabling grabbing somewhat sacrifices fairness, the effects are insignificant in practice. Moreover, unlike most of the previous results, the *CASN* operation does not require the list of addresses to be sorted before calling, and the *Read* operation can read the current value without applying helping when the word to be read is within an ongoing transaction. Experiments using micro-benchmarks varying important parameters in three dimensions have been performed on two multiprocessor platforms. The results show similar performance as the lock-free algorithm by Harris et al. for most scenarios, and significantly better performance on scenarios with very high contention. This is altogether extraordinary good performance considering that the new algorithm is wait-free.

**Keywords:** wait-free, multi-thread, compare-and-swap, real-time

## 1. Introduction

In multi-thread programming it can sometimes be useful to be able to update a set of shared variables simultaneously. An example is when one is designing dynamic data structures and needs to update several pointers at the same time, or when conducting scientific calculations in a graph structure and needs to update several nodes simultaneously. To do this accurately a transaction is needed, and as the means to do this is not normally directly supported by the system, special constructs are needed. A common way to implement a multi-variable transaction feature is to use mutual exclusion; the concurrent execution is serialized by the use of locks such that only one thread is updating the variables at a time. However, locks inherently limit the achievable parallelism and therefore also significantly degrade the overall performance.

Moreover, mutual exclusion causes blocking and can consequently incur serious problems as deadlocks, priority inversion or starvation. These problems are especially important for real-time systems, and efficient solutions only exist for uni-processor systems [1]. Some researchers have addressed these problems by introducing non-blocking synchronization algorithms, which are not based on mutual exclusion. Lock-free algorithms are non-blocking, and guarantee that always at

least one operation can progress, independently of the actions taken by the concurrent operations. Thus, lock-free algorithms can potentially incur starvation, although the other problems with mutual exclusion are avoided. Wait-free [2] algorithms are lock-free, and moreover guarantee that all operations can finish in a finite number of their own steps, regardless of the actions taken by the concurrent operations. In practice, often due to the extensively needed synchronization, wait-free algorithms are magnitudes more complex to design and offer significantly lower performance than corresponding lock-free. Consequently, wait-free algorithms are normally of special interest to real-time systems, whereas lock-free algorithms' primary benefit is of performance.

The design of non-blocking algorithms benefits from the fact that the hardware of contemporary shared memory systems supports atomic transactions on a single memory word, e.g. single-word compare-and-swap (CAS). The CAS operation can conditionally update a memory word such that the new value is written only if the old value matches a given one. It has been shown that this hardware primitive is universal and thus can implement any shared data structure in a non-blocking manner. Consequently, it is possible to construct a multi-word compare-and-swap operation (*CASN*) by the use of CAS and other hardware primitives. The *CASN* operation conditionally updates a set of memory words to a new set of values given that the words currently match a given set of values.

Several non-blocking algorithms suitable for and actual implementations of *CASN* have appeared in the literature. From a historic perspective, the focus on these papers has changed with time from the theoretic side to more practical implementations. Israeli and Rappoport [3] presented a disjoint-access-parallel algorithm. Anderson and Moir [4] presented a wait-free algorithm. Shavit and Touitou [5] generalized the concept into software transactional memory<sup>1</sup>. Attiya and Dagan [6], [7] and Afek et al. [8] presented algorithms focusing on aspects for general multi-object operations. Moir [9] presented a special purpose algorithm which is conditionally wait-free. These algorithms have in common to use recursive helping techniques and using large parts of the memory words for the synchronization. Harris et al. [10] presented a lock-free algorithm that improves performance significantly and increased the useable part of the memory words by using separate descriptor

<sup>1</sup>Software transactional memory (STM) has spawned a new area of research for generalized concurrent programming, which is out of scope for this paper.

structures for the synchronization information. Ha and Tsigas [11] presented a lock-free algorithm that focused on improving the handling of conflicts by only releasing the necessary amount of words in a reactive manner. In this paper we will primarily focus on those results that are practical and generally applicable on contemporary systems.

This paper presents a new algorithm that implements a wait-free and linearizable multi-word compare-and-swap feature. The algorithm uses greedy helping techniques in order to limit the amount of recursive helping, takes benefits from descriptors to allow a larger part of the memory word to be useable, uses grabbing in a deterministic scheme to resolve conflicts, and allows the given list of addresses to be unsorted when calling the *CASN* operation.

The rest of the paper is organized as follows. Section 2 describes the system requirements and in Section 3 the new algorithm is described. Section 4 sketches the algorithm's correctness proof. In Section 5, some benchmark experiments are shown. Finally, Section 6 concludes this paper.

## 2. Hardware Synchronization

The shared memory system should support atomic<sup>2</sup> read and write operations of single memory words, as well as stronger atomic primitives for synchronization. In this paper we use the Fetch-And-Add (FAA), Compare-And-Swap (CAS) and the Swap (SWAP) atomic primitives; see Program 1 for a description. These read-modify-write style of operations are available on most common architectures or can be easily derived from other synchronization primitives [12] [13].

---

**Program 1** The Fetch-And-Add (FAA), Compare-And-Swap (CAS) and Swap (SWAP) atomic primitives.

---

**procedure** FAA(address: **pointer to word**, number: **integer**)

**atomic do**

*\*address* := *\*address* + number;

**function** CAS(address: **pointer to word**, oldvalue: **word**, newvalue: **word**): **boolean**

**atomic do**

**if** *\*address* = oldvalue **then**

*\*address* := newvalue;

**return true;**

**else return false;**

**function** SWAP(address: **pointer to word**, newvalue: **word**): **word**

**atomic do**

oldvalue := *\*address*;

*\*address* := newvalue;

**return** oldvalue;

---

## 3. New Algorithm

In order to perform multi-word compare-and-swap as well as other operations on the shared memory words, all accesses

<sup>2</sup>either natively or by employing adequate memory barrier instructions

to the affected words have to be done through newly defined operations. In this paper we are defining the *Read* and the *CASN* operations:

**function** Read(address: **pointer to word**): **integer**

**function** CASN(addresses: **array of pointer to word**, olds: **array of integer**, news: **array of integer**): **boolean**

In an abstract sense, the overall algorithm for performing the *CASN* operation is to:

- 1) lock all the affected memory words.
- 2) check the contents of the memory words and perform the conditional update.
- 3) unlock all the affected memory words.

The information of a word's lock-status is naturally stored within the memory word itself (an interesting alternative would be a hash-table). Moreover, in order to be wait-free, we need to be able to perform step 2 in one atomic step, correctly read the current value even though words might be locked, and perform helping of operations in the middle of steps 1-3 while other operations are pending. To meet these requirements we are using descriptors, see Figure 1, that keep all necessary information about a *CASN* operation in progress, and locking is done by replacing the value with a pointer to the appropriate descriptor. As the descriptor keeps a single variable that indicates the status of the whole *CASN* operation, it is possible to update this atomically using CAS.

The status of the descriptor is stored in a single memory word and can be one of STATUS\_TRYING(seq), STATUS\_GIVE(descriptor), STATUS\_FAILED, or STATUS\_SUCCESS. The descriptor also keeps information about the involved addresses, requested old and new values, as well as information about the owning thread. By first checking the memory word and then checking the status (depending on which the interpreted value is either the old or new value) of the possibly corresponding descriptor, it is possible for other concurrent operations to read the current value of a memory word involved in a *CASN* operation that is in some intermediate step. In order to distinguish between the actual value and a reference to a descriptor one bit is used, leaving 31 bits on a 32-bit memory word for the value.

The concurrent *CASN* operations will occasionally operate on the same memory words, as can be seen in Figure 2, and conflicts in the locking step will eventually occur. The overall idea used in this paper is first let the whole system of concurrent operations to stabilize by performing helping of the involved operations until no thread can lock any more words. After that, the conflicting words are identified and by a deterministic rule it is decided which thread should grab the necessary locked words from the other thread. In this way, all conflicts will eventually be resolved, even if involved by more than two threads and interchangeably.

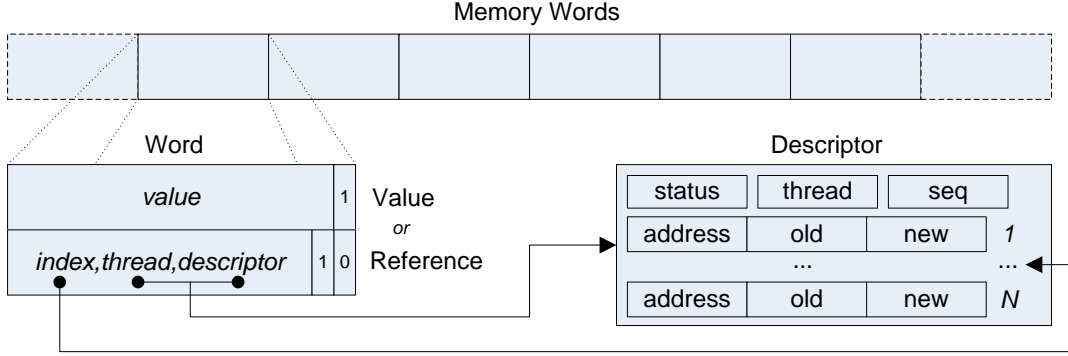


Fig. 1: The CASN descriptor and its references from memory words.

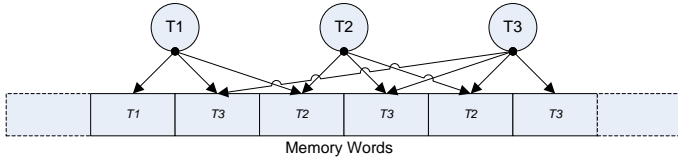


Fig. 2: Example of a conflict between three threads, where all available words have been locked and none of the threads can proceed further. This conflict can be resolved deterministically by changing the ownership of the locks appropriately.

### 3.1 Wait-free reading

Both the *Read* and *CASN* operations involve the subtask of interpreting the current value of a memory word, concurrently with other operations. Thus, in order for the operations to be designed wait-free, we need to be able to read the current value of a memory word in a wait-free manner, avoiding any (unlimited) repetition in case the memory word was concurrently updated. Moreover, as this reading is done extensively, being part of step 2 of the overall *CASN* algorithm, it should be done as efficiently as possible. Many of the previous results, e.g. [10], [11], need to apply helping of concurrent *CASN* operations while reading a memory word currently involved in an undecided *CASN* in order to correctly interpret the word’s current value. Thanks to wait-free dereferencing and direct index-to-descriptor lookup, see Figure 1, this algorithm can avoid this helping.

For the purpose of wait-free dereferencing we are using the same idea as presented in [14], which is based on reference counting (thus adding the *refcount* field to the descriptor structure) to make sure that shared data structures like the descriptors can be safely accessed while the corresponding memory concurrently being reclaimed. The idea benefits from the fact that a memory word can store integers as well as pointers. As can be seen in Program 2, each thread has a set of shared variables used for announcing their intentions. The function *WFRead* returns the contents of the read memory word, and if it was a reference it also returns a pointer to the corresponding descriptor. Before actually reading the memory

word in line W4, the address of it is stored in one of the slots of the *annReadAddr* array in line W3. If the value read is a reference the address to the corresponding descriptor is fetched and its reference count increased in lines W5-W7. In case there was a concurrent update of the read memory word after line W3 and the memory of the descriptor possibly being reclaimed for other use, the word in the *annReadAddr* array has also been concurrently changed<sup>3</sup> in line W8-W9. If so, the new value or descriptor reference is fetched in lines W10-W14. This procedure works by forcing the concurrent threads to follow a certain rule whenever performing reclamation. Before reclaiming the memory of a descriptor (suitably after a *CASN* operation has finalized), all of the threads’ *annReadAddr* variables have to be scanned to see if the announced addresses corresponding to any of the addresses in the descriptor to be reclaimed. If so, the corresponding *annReadAddr* is updated with a fresh value or descriptor which adheres to the current value of the memory word. Initially, all memory words need to be initialized with contents that can be properly interpreted as values.

### 3.2 HelpCompareAndSwap

To enable scalable performance with an increasing number of threads, the algorithm needs to be as disjoint-access-parallel as possible, e.g., two *CASN* operations on a disjoint set of memory words should not synchronize. For the algorithm to be wait-free, every operation should eventually terminate if executing a finite number of steps. In general, this implies the need for helping concurrent operations that are in some type of conflict, e.g., in a way such that all threads trying a *CAS* for a certain update will eventually succeed. On the other hand, for performance reasons, helping should be avoided as much as possible as it essentially means having multiple threads perform each thread’s work. A greedy approach would be to have each

<sup>3</sup>Before possibly reclaiming a descriptor, helping of announced read operations are performed, and the result of helping is returned by updating (*CAS*) the corresponding announcement (*annReadAddr*). While performing the helping, i.e. reading the address, *annBusy* of the corresponding index is set to true in order avoid late helping, in the same manner as described in [14].

---

**Program 2** Procedure for performing wait-free reading of a memory word's contents.

---

*For each thread, accessible by all*

annBusy: **array of boolean**; // Initially set to false

annIndex: **integer**;

annReadAddr: **array of word**; // Initially set to  $\perp$

```

function WFRRead(address: pointer to word): (word
, pointer to descriptor)
W1  Choose index such that annBusy[index] = false .
W2  annIndex := index;
W3  annReadAddr[index] := address;
W4  ref := *address;
W5  if IS_REFERENCE(ref) then
W6    desc := GET_DESCRIPTOR(ref);
W7    FAA(&desc.refcount,1);
W8  refRep := SWAP(&annReadAddr[index],  $\perp$ );
W9  if refRep  $\neq$  address then
W10   if IS_REFERENCE(ref) then
W11     FAA(&desc.refcount,-1);
W12   ref := refRep;
W13   if IS_REFERENCE(ref) then
W14     desc := GET_DESCRIPTOR(ref);
W15 return (ref, desc);

```

---

thread perform helping only if it is needed for *that* thread to progress. An observation of the normally expected execution behavior of concurrent *CASN* operations, is that if one *CASN* succeeds it typically means that all concurrent *CASN* operations that are in conflict will terminate with failure (as the updated memory words no longer match the expected values). Thus, assume that we could achieve a conflict resolution scheme such that if at least one thread executes its steps, one of the conflicting *CASN* operations will eventually succeed. Then all other conflicting threads will also eventually terminate and in total the operations will be wait-free. However, there are unlikely, but still possible scenarios, e.g. one thread executing a series of successful  $CAS(x,0,0), CAS(x,0,0), \dots$  concurrently with another thread issuing  $CAS(x,0,1)$  which unfortunately never wins the conflict, where one or more threads consequently starves. Hence, as this kind of scenario is unlikely in itself and even more unlikely to repeat infinitely, we define the following assumption:

**Assumption 1:** When used in the system, the successful *CASN* operations are always updating the memory words to new values that are different compared to the old, or at least there is a limited series of updates to same values in the concurrent execution history.

The procedure *HelpCompareAndSwap* is used by both the issuing *CASN* operation and by operations that need to help that *CASN* in order to continue with its own operation. In order to avoid recursive helping of the same threads, the procedure keeps a local set data structure, *avoidList*, that keeps track of which previous threads that are in the same recursive call-chain of the current *HelpCompareAndSwap* procedure.

The *HelpCompareAndSwap* procedure starts with the status of *STATUS\_TRYING* and keeps on trying to lock all the memory words until either it is not possible to proceed further with this descriptor (due to conflicts which could not be resolved without helping some thread in the *avoidList*) or the status of the descriptor has finalized to either *STATUS\_SUCCESS* or *STATUS\_FAILED*.

While the status is *STATUS\_TRYING*(seq), the procedure first tries to lock (using CAS) all words with the correct value (matching with the requested old) that are either free or belonging to a descriptor which has finalized. If a word is found having the wrong value, the status is updated (with CAS) to *STATUS\_FAILED*. If no more words can be locked and there are still words remaining before success, the procedure now tries to resolve the conflict as follows:

- If the conflicting thread that owns the conflicting word belongs to the *avoidList*, this word is skipped (which means that this call of the procedure cannot finalize the descriptor) and the procedure keeps on with the next conflicting word.
- If the helped thread has a lower id than the conflicting thread, the procedure tries to start grabbing by trying to update (with CAS) the status of the conflicting descriptor from *STATUS\_TRYING*(seq2) to *STATUS\_GIVE*(helped descriptor). If this succeeds it then grabs (with CAS) all of the words needed from the conflicting thread and then updates the status of the conflicting descriptor to *STATUS\_TRYING*(seq2+1) (the increased sequence number is required for making the conflicting thread aware that words have been lost). Otherwise it recursively calls the *HelpCompareAndSwap* procedure for the conflicting descriptor with the currently helped thread added to the *avoidList*.
- If the helped thread has higher id than the conflicting thread, the procedure calls the *HelpCompareAndSwap* procedure for the conflicting descriptor with the currently helped thread added to the *avoidList*. It will then try to change the status of the helped descriptor and possibly give away the required words in the same (although inversely in the respect of thread id and descriptors) manner as the above case.

While the status is *STATUS\_GIVE*(descriptor2), the procedure will give (with CAS) all of the words needed by the conflicting thread (given by descriptor2) that is currently locked by the helped thread. It will then update the status of the helped descriptor to *STATUS\_TRYING*(seq+1) (where *STATUS\_TRYING*(seq) was the status before it became *STATUS\_GIVE*).

If the status becomes either *STATUS\_SUCCESS* or *STATUS\_FAILED*, the procedure will perform a clean-up on all locked memory words, updating them (with CAS) to pure values and thus removing the reference to the descriptor, and then terminate.

### 3.3 Read and CASN operations

The design of the *Read* operation follows directly by the use of the *WFRead* function. If the content at the address was a value, the significant bits are returned. Otherwise, the word index (according to the *CASN* operation) is extracted, and depending on the status of the descriptor, either the requested old or new value at the corresponding index in the descriptor is returned.

The design of the *CASN* operation, is to simply initialize a new (which needs to be dynamically allocated and reclaimed in a wait-free manner [14], from either a shared or local memory pool) descriptor with the arguments. The *HelpCompare-AndSwap* procedure is then called on the descriptor together with an empty *avoidList* (such that all conflicts will be resolved in this call). Depending on the resulting status of the descriptor the operation returns either true or false.

### 3.4 Algorithm extensions

As the conflicts are deterministically resolved by using the threads' ids, the algorithm is not fair. Although this is no problem in practice, it is possible to improve the fairness. Either the thread ids can be cycled in a round-robin manner for every *CASN* operation issued, or the deterministic conflict resolution ordering are dynamically changed in run-time (however, the order needs to avoid the possibility of bouncing).

The limitation of not being able to use all bits of the memory word for values, can be overcome by replacing the value bits with a pointer to a separate memory block (which is dynamically allocated and reclaimed by a separate scheme) of arbitrary size; an interesting technique also used in [15].

The algorithm can be made purely wait-free (e.g. avoiding Assumption 1) by allowing a *CASN* operation that have failed to lock or been forced to release any word, to then announce its operation. Other threads are then forced to initially check this announcement and help correspondingly.

## 4. Correctness

In this section we sketch the correctness proof by showing the wait-free and linearizability properties of the algorithm.

**Lemma 1:** The *Read* operation is wait-free.

*Proof:* As the algorithm steps of the used *WFRead* and the remaining steps of the *Read* operation contains no loops, it follows that the operation must terminate in a finite number of executed steps. ■

**Lemma 2:** The *CASN* operation is wait-free.

*Proof:* Firstly, all reads of memory words are clearly wait-free, as *WFRead* is used. The only unbounded loop is the main loop of the descriptor status, which will terminate if either all words are successfully locked or there was a word not matching the requested old value. It will retry if there was a concurrent update to any of the words it tried to lock or if the status is changed. If there are no conflicts with concurrent *CASN* operations, there will not be any concurrent updates of the corresponding memory words, and the operation will

consequently eventually terminate. If there was a conflict, it is either the case that the conflicting words are grabbed or given. If they are grabbed, the operation will clearly eventually terminate. If they are given, the conflicting operation will be helped. The helping will terminate in a finite number of steps as any further recursive helping is limited by the number of threads and subsequent conflicts will be resolved deterministically. The helped operation will succeed and according to our assumption, the words are updated with new values of which at least one is not matching the requested old values of this operation, and consequently this operation will eventually terminate. ■

**Definition 1:** The value of a memory word is interpreted depending on its contents as follows. If it is marked as a value, the interpreted value is the remaining integer bits. Otherwise, if the status of the corresponding descriptor is success, the interpreted value is the requested new value of the corresponding address, and otherwise it is the requested old value.

**Lemma 3:** The *Read* operation takes effect atomically.

*Proof:* If the read content of the memory word was marked as a value, the operation takes effect at the read sub-operation. If the content was marked as a reference, there might have been a concurrent update of the descriptor's status before the status was read. If there was a concurrent update of the status to success, the operation takes effect at the concurrent CAS sub-operation (at this point, the memory word must still be locked and consequently follows Definition 1). Otherwise, the operation takes effect at the read sub-operation of the memory word. ■

**Lemma 4:** The *CASN* operation takes effect atomically.

*Proof:* A *CASN* operation that succeeds also updates the corresponding values. Consequently, it should take effect when the update takes effect according to Definition 1, which is when the descriptor's status is changed to success with the CAS sub-operation. When a *CASN* operation fails, there is one memory word not matching the requested old values. Consequently, it should take effect when the non-matching word is read according to how the *Read* operation takes effect. ■

**Definition 2:** In order for an implementation of a shared concurrent data object to be linearizable [16], for every concurrent execution there should exist an equal (in the sense of the effect) and valid (i.e. it should respect the semantics of the shared data object) sequential execution that respects the partial order of the operations in the concurrent execution.

**Theorem 1:** The algorithm implements a wait-free (under given assumptions about execution history) and linearizable multi-word compare-and-swap functionality.

*Proof:* Following Lemmas 1 and 2 given Assumption 1, the algorithm is wait-free. According to Lemmas 3 and 4, the *Read* and *CASN* operations take effect atomically at one statement that is executed within their invocations. Consequently, these operations are linearizable according to the given definitions. ■

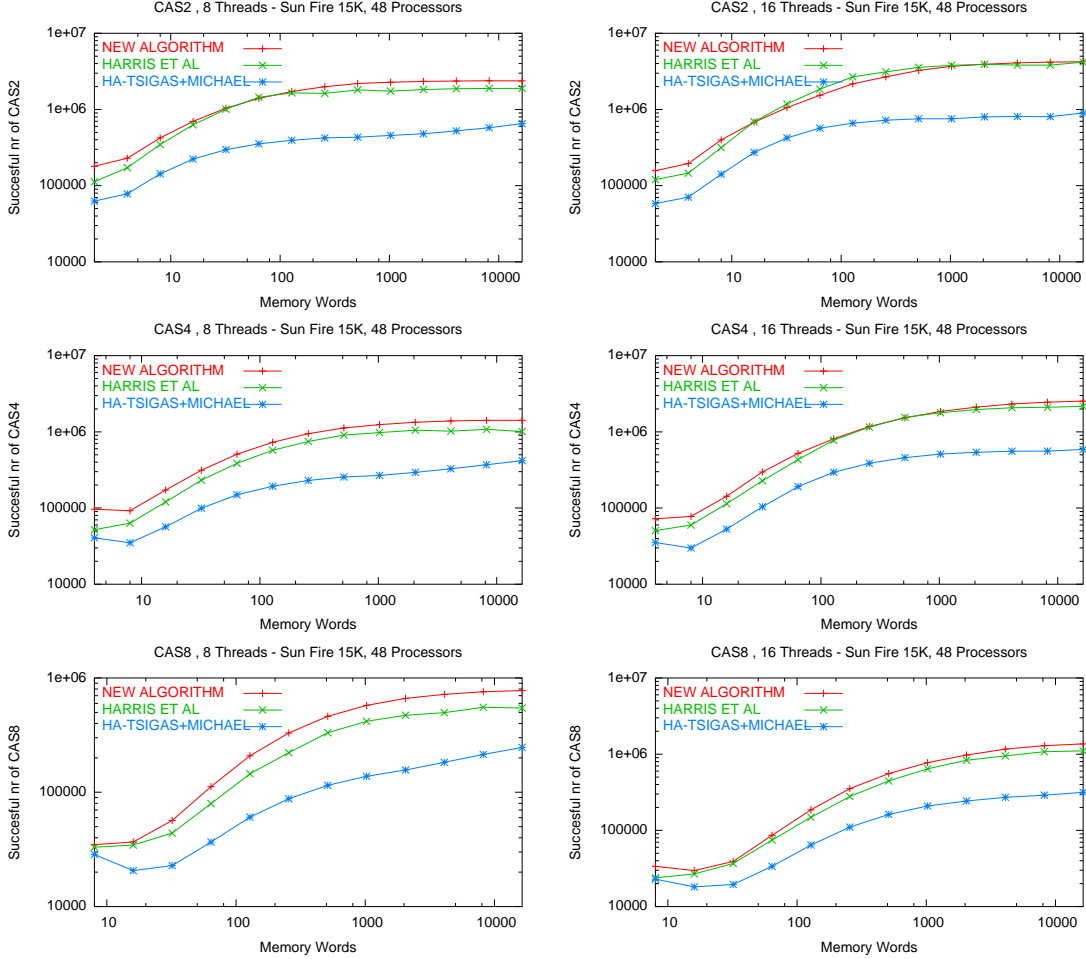


Fig. 3: Experiments on a 48-way Sun Fire 15K system.

## 5. Experiments

We have conducted a number of experiments in order to examine the behavior of the algorithm on contemporary multiprocessors in the respect of the number of words, number of threads, and level of contention. The experiments are performed in the form of a micro-benchmark. In this benchmark, each thread is repeatedly trying to atomically increment a randomized (for every repetition, determined off-line before starting each benchmark so that all implementations can use the very same random patterns) set of words by first reading them by using the *Read* operation and then updating them by using the *CASN* operation. For checking correctness, in every repetition the result of the *CASN* is noted down in local memory, and the whole history is calculated after the benchmark and compared with the concurrent execution. The repetition continues for 5 seconds and then the total (by all threads) number of successful *CASN* operations is estimated.

The benchmarks are executed with varying parameters in three dimensions. The number of words is varied between 2, 4, 8, or 16 words. The number of threads is varied between

8, 16, or 32 threads. The level of contention is varied by varying the total number of memory words to randomly choose from between 2, 4, 8, 16, 32, ..., up to 16384 memory words. Besides the algorithm in this paper, the experiments were also conducted with the algorithms by Harris et al. [10], and by Ha and Tsigas [11]. For the memory management of the implementations of this algorithm and by Harris et al., the memory for the descriptors are pre-allocated in local pools for each thread with reclamation managed by simple reference counting [17]. As the algorithm by Ha and Tsigas requires ideal LL/VL/SC atomic operations<sup>4</sup>, this was implemented using the algorithm by Michael [15]. The number of useable bits of the memory word for actual values in each implementation was 31, 30, and 32 respectively. All algorithms were implemented in C and Assembler (for atomic primitives and memory barriers), compiled with highest optimization.

We have performed our experiments on two multiprocessor

<sup>4</sup>Note that the algorithm by Ha and Tsigas performs significantly better with the LL/SC implementation by [12], however then leaving very few bits useable for values unless 64-bit CAS is used on 32-bit systems.

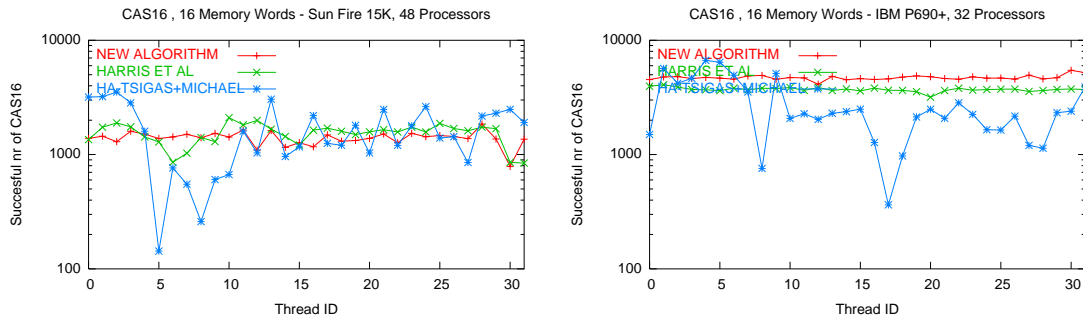


Fig. 4: Experiments regarding the experienced fairness.

platforms; a Sun Fire 15K system running Solaris 9 with 48 Ultrasparc III processors of 900 MHz each, and also on an IBM p690+ Regatta system running AIX with 32 processors of 1.7 GHz each. Results from a subset of the experiments are shown in Figure 3 for the Sun platform, and for the IBM platform similar results were observed. In Figure 4 is shown results in respect of fairness for each involved thread under maximum contention, for respective multiprocessor platform.

The results show in most of the scenarios similar performance of the new wait-free algorithm compared to the lock-free algorithm by Harris et al, as well as significantly worse in some scenarios with medium contention. However, in scenarios with maximum contention (equal number of words as number of memory words) it performs significantly better in majority of the experiments. Moreover, it seems in practice that fairness is not worse than for the other algorithms compared. The advantage in high contention is likely to be thanks to the efficient conflict resolution by grabbing and the greedy helping that enables operations to steal memory words of other just recently terminated operations without having to wait for them to clean up.

## 6. Conclusions

We have presented a new algorithm for implementing a multi-word compare-and-swap functionality supporting the *Read* and *CASN* operations. The algorithm is wait-free under reasonable assumptions on execution history and benefits from new techniques to resolve conflicts between operations by using greedy helping and grabbing. Moreover, unlike most of the previous results, the *CASN* operation does not require the list of addresses to be sorted prior to the call.

Experiments have been conducted on two multiprocessor platforms. Results show similar performance as the algorithm by Harris et al. for most scenarios, and significantly better performance on scenarios with very high contention.

Interesting future work is to investigate the usefulness of the new algorithm for implementing dynamic wait-free data structures, improving fairness while preserving the overall performance, and to improve the performance of the underlying memory management.

## Acknowledgement

Part of this work was carried out under the HPC-EUROPA project (RII3-CT-2003-506079), Infrastructure Action under the FP6 “Structuring the European Research Area” program.

## References

- [1] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.
- [2] M. Herlihy, “Wait-free synchronization,” *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, pp. 124–149, Jan. 1991.
- [3] 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*, Aug. 1994.
- [4] J. H. Anderson and M. Moir, “Universal constructions for multi-object operations,” in *Proceedings of the 14th Annual ACM Symposium on the Principles of Distributed Computing*, Aug. 1995.
- [5] N. Shavit and D. Touitou, “Software transactional memory,” in *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. ACM Press, 1995, pp. 204–213.
- [6] H. Attiya and E. Dagan, “Universal operations: Unary versus binary,” in *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, 1996, pp. 223–232.
- [7] —, “Improved implementations of binary universal operations,” *Journal of the ACM*, vol. 48, no. 5, pp. 1013–1037, 2001.
- [8] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou, “Disentangling multi-object operations,” in *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, 1997, pp. 111–120.
- [9] M. Moir, “Transparent support for wait-free transactions,” in *Proceedings of the 11th International Workshop on Distributed Algorithms*, Sept. 1997.
- [10] T. Harris, K. Fraser, and I. Pratt, “A practical multi-word compare-and-swap operation,” in *Proceedings of the 16th International Symposium on Distributed Computing*, 2002.
- [11] P. H. Ha and P. Tsigas, “Reactive multi-word synchronization for multi-processors,” *Journal of Instruction-Level Parallelism*, vol. 6, Apr. 2004.
- [12] M. Moir, “Practical implementations of non-blocking synchronization primitives,” in *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, Aug. 1997.
- [13] P. Jayanti, “A complete and constant time wait-free implementation of CAS from LL/SC and vice versa,” in *DISC 1998*, 1998, pp. 216–230.
- [14] H. Sundell, “Wait-free reference counting and memory management,” in *Proceedings of the 19th International Parallel & Distributed Processing Symposium*. IEEE, Apr. 2005.
- [15] M. M. Michael, “Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS,” in *Proceedings of the 18th International Conference on Distributed Computing*, 2004, pp. 144–158.
- [16] M. Herlihy and J. Wing, “Linearizability: a correctness condition for concurrent objects,” *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.
- [17] J. D. Valois, “Lock-free data structures,” Ph.D. dissertation, Rensselaer Polytechnic Institute, Troy, New York, 1995.