Technical Report no. 2003-06

Fast, Reactive and Lock-free Multi-word Compare-and-swap Algorithms

Phuong Ha-Hoai **Philippas Tsigas**

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2003

Technical Report in Computing Science at
Chalmers University of Technology and Göteborg University

Technical Report no. 2003-06
ISSN: 1650-3023

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2003

Abstract

Shared memory multiprocessors typically provide a set of single-word compare-and-swap-*like hardware primitives to support synchronization. Although these are conceptually powerful enough to support higher-level synchronization, from the programmer's point of view they are not as useful as their generalizations to the* multi-word *objects.*

This paper presents two fast and reactive lock-free multi-word *compare-and-swap algorithms. The algorithms dynamically measure the level of contention and automatically react to guarantee good performance. Experiments on a SGI-Origin2000 multiprocessor show that our algorithms react fast according to the contention variations and perform two to nine times faster than the best-known alternatives.*

1 Introduction

Synchronization is an essential point of hardware/software interaction. On one hand, programmers of parallel systems would like to be able to use high-level synchronization operations. On the other hand, the systems can support only a limited number of hardware synchronization primitives. Typically, the implementation of the synchronization operations of a system is left to the system designer, who has to decide how much of the functionality to implement in software in system libraries. There has been a considerable debate about how much hardware support and which hardware primitives should be provided by the systems.

Consider the multi-word compare-and-swap operations (CASNs) that extend the single-word compare-and-swap operations from one word to many. A single-word compare-and-swap operation (CAS) takes as input three parameters: the address, the old value and the new value of a word, and atomically updates the contents of the word if its current value is the same as the old value. Similarly, a N-word compare-and-swap operation takes the addresses, the old values and the new values of N words, and if the current contents of these N words all are the same as their respective old values, the CASN will update the new values to the respective words atomically. Otherwise, it will fail. Because of this powerful feature, CASN makes the design of concurrent data objects much more effective and easier than the single-word compare-and-swap [5][6][7]. On the other hand most multiprocessors support only single word compare-and-swap or compare-and-swap-like operations e.g. Load-Link/Store-Conditional in hardware.

As it is expected, many research papers implementing the powerful CASN operation have appeared in the literature [1][2][8][12][14][16]. Typically, in a CASN implementation, a CASN operation tries to lock all words it needs one by one. During this process, if a CASN operation is blocked by another CASN operation, then the process executing the blocked CASN will help the blocking CASN. Even though most of the CASN designs use the helping technique to achieve the lock-free or wait-free property, the helping strategies in the designs are different. In the *recursive helping policy* [1][8][12], the CASN operation, which has been blocked by another CASN operation, never releases the words it has acquired even though many other not conflicting CASNs might have been blocked on these words. On the other hand, in the *software transactional memory* [14][16] the blocked CASN operation immediately releases all words it has acquired regardless of whether there is any other CASN in need of these words at that time. In low contention situations, the release of all words acquired by a blocked CASN operation will only increase the execution time of this operation without helping many other processes. Moreover, in any contention scenario, if a CASN operation is close to acquiring all the words it needs, releasing all its acquired words will not only increase significantly its execution time but also increase the contention in the system when it tries to acquire these words again. The disadvantage of both strategies is that both of them are not adaptable to the different access patterns of the memory that different CASNs can trigger as well as frequent variations of the contention on each word of shared data. This can actually have a large impact on the performance of these implementations.

The idea behind the work described in this paper is that giving the CASN operation the possibility to adapt its helping policy to variations of contention can have a large impact on the performance of a CASN implementation in most contention situations. Of course, dynamically changing the behaviour of the protocol comes with the challenge of performance. The overhead that the dynamic mechanism will introduce should not exceed the performance benefits that the dynamic behaviour will bring.

The rest of this paper is organized as follows. We give a brief problem description, summarize the related work and give more detailed description of our contribution in Section 2. Our algorithms are described in Section 3. The correctness proof of our algorithms is presented in Section 4. In Section 5 we present the evaluation of the performances of our CASN algorithms and compare them to the best known alternatives, which also represent the two helping strategies mentioned above. Finally, Section 6 concludes the paper.

2 Problem Description, Related Work and Our Contribution

Concurrent data structures play a significant role in multiprocess systems. To ensure consistency of a shared data object in a concurrent environment, the most common method is to use mutual exclusion, i.e. some form of locking. Mutual exclusion degrades the system's overall performance as it causes blocking, i.e. other concurrent operations can not make any progress while the access to the shared resource is blocked by the lock. Using mutual exclusion can also cause deadlocks, priority inversion and even starvation.

To address these problems, researchers have proposed non-blocking algorithms for shared data objects. Non-blocking methods do not involve mutual exclusion, and therefore do not suffer from the problems that blocking can cause. Non-blocking algorithms are either lock-free or wait-free. Lock-free implementations guarantee that regardless of the contention caused by concurrent operations and the interleaving of their sub-operations, always at least one operation will progress. However, there is a risk for starvation as the progress of other operations could cause one specific operation to never finish. This is although different from the type of starvation that could be caused by blocking, where a single operation could block every other operation forever, and cause starvation of the whole system. Wait-free [11] algorithms are lock-free and moreover they avoid starvation as well, in a wait-free algorithm every operation is guaranteed to finish in a limited number of steps, regardless of the actions of the concurrent operations. Non-blocking algorithms have been shown to be of big practical importance [18, 19], and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [17].

The main problem of lock/wait-free concurrent data structures is that many processes try to read and modify the same portions of shared data at the same time and the accesses must be atomic to one another. That is why a multi-word compare-and-swap operation is so important for such data structures.

Herlihy proposed a methodology for implementing concurrent data structures where interfering processes is prevented by generating a private copy of the portion changed by each process [10]. The disadvantages of Herlihy's methodology are the high cost for copying large objects and not disjoint-access-parallel, which means the processes accessing no common portion should progress in parallel.

Barnes [3] later suggested a *cooperative technique* which allows many processes to access the same data structure concurrently as long as they write down exactly what they are doing. Before modifying a portion of the shared data, a process p_1 checks whether the portion is used by another process p_2. If yes, process p_1 will cooperate to complete the work of process p_2.

Israeli and Rappoport transformed this technique into one more applicable in practice in [12], where the concept of disjoint-access-parallel was given. All processes try to lock the whole portions of shared data they need before writing the new values to the portions one by one. Each portion has field *owner* to inform which process is the owner of the portion at that time.

Harris, Fraser and Pratt [8] aiming to reduce per-word space overhead eliminated the owner field. They exploited the data word for containing a special value, a pointer of CASNDescriptor, to inform which process is the owner of the data word. However, the vague point in the paper is how to solve the memory management problem in their method.

A wait-free multi-word compare-and-swap was introduced by Anderson and Moir in [1]. However, the disadvantage of the so called *cooperative technique*, which was used in this paper, is that the process, which is blocked by another process, does not release the words it owns when it helps the blocking process, even though many other processes blocked on these words can progress if these words are released. This *cooperative technique* uses a *recursive helping policy*, and the time taken for a blocked process p_1 to help another process p_2 may be long. Moreover, the longer the response time of p_1, the bigger the number of processes blocked by p_1. The processes blocked by p_1 will first help process p_1 and then continue to help process p_2 even when they and process p_2 access disjoint parts of the data structure. This problem will be solved if process p_1 does not conservatively keep its words and releases them while it is helping the blocking process p_2.

Shavit and Touitou realized the problem above and presented the *software transactional memory* (STM) in [16]. In STM, a process p_1 that was blocked by another process p_2 releases the words it owns immediately before helping blocking process p_2. Moreover, a blocked process helps at most a blocking process, so *recursive helping* does not occur. STM then was improved by Mark Moir [14], who introduced a design of a conditional wait-free multi-word compare-and-swap operation. An evaluating function passed to the CASN by the user will identify whether the CASN will retry when the contention occurs. Nevertheless, both STM and the improved version (iSTM) also have the disadvantage that the blocked process releases the words it owns regardless of the contentions on the words. That is even if there is no other process requiring the words at that time, it still releases the words, and after helping the blocking process, it will have to compete with other processes to acquire the words again. Moreover, even if a process acquired the whole words it needs except for the last one owned by another process, it still releases all the words impatiently and then starts from scratch. In this case, it should realize that not many processes require the words and it is nearly successful, so try to keep the words as in *cooperative technique*.

2.1 Synchronization primitives

In this subsection, we describe the synchronization primitives related to our reactive multi-word compare-and-swap algorithms. The definitions of the primitives are described in Figure 1.

---

fetch_and_add(m, v)
$atomically$ {
  $oldm \leftarrow m$;
  $m \leftarrow m + v$;
  return$(oldm)$
  }

compare_and_swap(m, old, new)
$atomically$ {
  if$(m = old)$
    $m \leftarrow new$;
    return$(true)$;
  else return$(false)$;
  }

LL(x)\{ $return\ the\ value\ of\ x\ such\ that\ it\ may\ be\ subsequently\ used\ with$ **SC** \}.

VL(x)
$atomically$ {
  if $(no\ other\ process\ has\ written\ to\ x\ since\ the\ last$ **LL**$(x))$
    return$(true)$;
  else return$(false)$;
  }

SC(x, v)
$atomically$ {
  if $(no\ other\ process\ has\ written\ to\ x\ since\ the\ last$ **LL**$(x))$
    $x \leftarrow v$; **return**$(true)$;
  else return$(false)$;
  }

Figure 1. Synchronization primitives

Practically, the SGI Origin2000 provides two hardware mechanisms for implementing synchronization primitives. The first one uses the *load-linked, store-conditional* instruction sequence of the MIPS instruction set and the second one uses the special *fetchop* hardware support for synchronization operations on uncached memory [20]. The latter is extremely fast for *read-modify-write* (RMW) instructions such as *fetch_and_increment, fetch_and_decrement*. The reason is that they require only two non-overlapped messages to and from the memory module storing the synchronization flag and do not create further network traffic because they bypass completely the cache-coherence protocol [15].

On the systems that support *LL/SC* but not *CAS* such as SGI Origin2000 or that support *CAS* but not *LL/SC* such as SUN machines, we can implement *LL/VL/SC* instructions by algorithms mentioned in [13].

2.2 Our Contribution

All available CASN implementations have their weak points. We realized that the weaknesses of these techniques came from their static helping policies. These techniques do not provide the ability to CASN operations to measure the contention that they generate on the memory words, and more significantly to reactively change their helping policy accordingly. We argue that these weaknesses are not fundamental and that one can in fact construct multi-word compare-and-swap algorithms where the CASN operations: i) measure in an efficient way the contention that they generate and ii) reactively change the helping scheme to help more efficiently the other CASN operations.

Synchronization methods that perform efficiently across a wide range of contention conditions are hard to design. Typically, "small" structures and "simple" methods fit better low contentions while "bigger" structures and more "complex" mechanisms can help to distribute processors to memory banks and thus alleviate memory contention. The key to our first algorithm is for every CASN to release the words it has acquired only if the average contention on the words becomes too high. This algorithm also favours the operations closer to completion. The key to our second algorithm is not to release all the words it owns at once but *just enough* so that most of the processes blocked on these words can progress. The performance evaluation on thirty processors of a SGI Origin2000 multiprocessor, which is presented in Section 5, comes in accord to our intuition. The new reactive CASN algorithms outperform significantly the best-known alternatives.

3 The Algorithms

In this section, we describe our reactive multi-word compare-and-swap implementations. In general, practical CASN operations are implemented by locking all the N words and then updating the new value to each word one by one. Only the

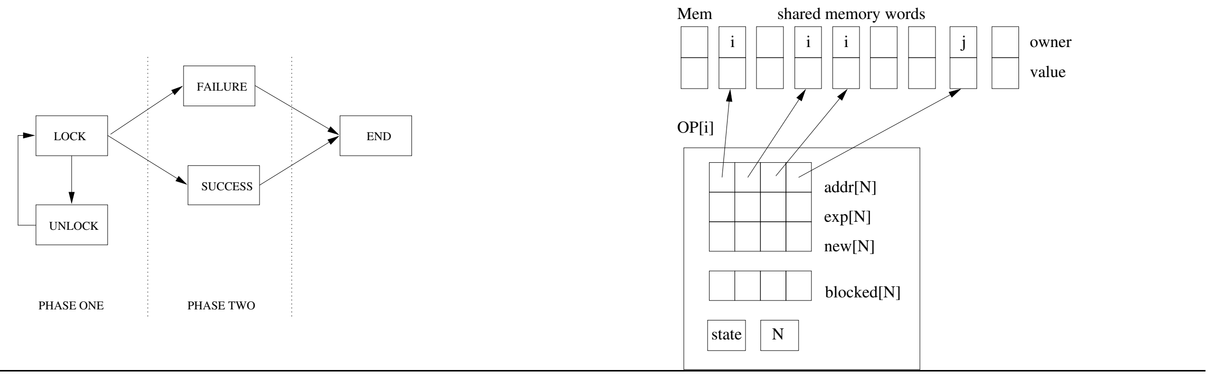


Figure 2. CASN states and CAS4 data structure

process having acquired the whole N words it needs can write the new values to the words, and the other processes, which are blocked, typically have to *help* the blocking process so that the lock-free feature is obtained. However, the helping scheme presented in [1][8][12][14][16] are based on different strategies that are described in Section 2. It should be mentioned here that different processes might require different number of words for their compare-and-swap operations and the number is not a fixed parameter.

3.1 First Reactive Scheme

The part of the operation that is of interest for our setting is the part that starts after a $CASN_i$ has acquired some words and then is blocked while trying to acquire a new word. The transaction of interest begins when $CASN_i$ is blocked on a word, and ends when it manages to acquire a new word, which could have been locked by $CASN_i$ before and then released by $CASN_i$. Our first reactive scheme decides whether and when the $CASN_i$ should release the words it has acquired by measuring the *contention* r_i on the words it has acquired, where $r_i = \frac{blocked_i}{kept_i}$ and $blocked_i$ is the number of processes blocked on the $kept_i$ words acquired by $CASN_i$. If the contention r_i is higher than a *contention threshold* R^*, process p_i releases all the words. One interesting feature of this reactive helping method is that it favours processes closer to completion as well as processes with a small number of conflicts. The *contention threshold* R^* is computed according to the *reservation price policy* (RPP) [4]. Our first reactive helping scheme obtains better performance than both *recursive helping policy* and *software transactional memory*. We will come back to evaluating performance of the method in Section 5. In the rest of the subsection, we will describe our algorithm in detail.

The variable $OP[i]$ described in Figure 2 is the shared variable that carries the data of $CASN_i$. It consists of four arrays with N elements each: $addr$, exp, new and $blocked$[1] that contain the addresses, the old values, the new values of the N words that need to be compared-and-swapped atomically and the number of processes blocked on the words, respectively. Each entry of the shared memory Mem, which is used by the real application, has two fields: the $value$ field and the $owner$ field. The $value$ field contains the real value of the word while the $owner$ field contains the identity of the CASN operation that has acquired the word. The $owner$ field needs $log_2 P$ bits, where P is the number of processes in the system.

Each CASN operation consists of two phases as described in Figure 2. The first phase has two states $Lock$ and $Unlock$, and it tries to lock all the necessary words according to our reactive helping scheme. The second one has also two states $Failure$ and $Success$. The second phase updates or releases the words acquired in the first phase. The pseudo-code of our algorithm is comprised by the procedures $Locking$, $Unlocking$, $Releasing$ and $Updating$ that are presented in Figure 3.

At the beginning, the CASN operation starts phase one in order to lock the N words. Procedure $Casn$ tries to lock the words it needs by setting the state of the CASN to $Lock$ (line 1 in $Casn$). Then, procedure $Help$ is called with four parameters: the identities of helping-process $helping$ and helped-CASN i, the position from which the process will help the CASN lock words pos, and version ver of current variable $OP[i]$.

- If the CASN operation manages to lock all the N words successfully, its state changes into $Success$ (line 7 in $Help$), then it starts phase two in order to conditionally write the new values to these words (line 10 in $Help$).

[1] In our implementation, the array $blocked$ is simple a 64-bit word. Thus, the whole array can be read in one atomic step. In general, the whole array can be read atomically by the snapshot technique

```
type word_type = record value; owner; end;
    para_type = record N: integer; addr: array[1..N] of *word_type ; exp, new: array[1..N] of word_type; /*N-word CAS*/
                state: {Lock, Unlock, Succ, Fail, Ends, Endf}; blocked[1..N] : 1..P; end; /*P: the number of processes*/
shared var Mem: set of word_type; OP: array[1..P] of para_type; Version: array[1..P] of unsigned long;
private var casn_l: array[1..P] of 1..P; l: of 1..P; /*keeping CASNs helped by the process*/
```

```
CASN(i)
begin
    OP[i].blocked := 0; OP[i].state := Lock;
    for j := 1 to P do casn_l[j] = 0;
    l := 1; casn_l[l] := i;
    Help(i, i, 0, Version[i]);
    return STATE[i];
end.

HELP(helping, i, pos, ver)
begin
start :
    state := LL(&OP[i].state);
    if (ver ≠ Version[i]) then return (Fail, nil);
    if (state = Unlock) then
      Unlocking(i);
      if (helping = i) then
        SC(&OP[i].state, Lock); goto start;
      else
        FAA(&OP[i].blocked[pos], −1);
        return (Succ, nil);
    else if (state = Lock) then
      result := Locking(helping, i, pos);
      if (result.kind = Succ) then
        SC(&OP[i].state, Succ);
      else if (result.kind = Fail) then
        SC(&OP[i].state, Fail);
      else if (result.kind = CB) then
        FAA(&OP[i].blocked[pos], −1);
        return result;
      goto start;
    else if (state = Succ) then
      Updating(i); SC(&OP[i].state, Ends);
    else if (state = Fail) then
      Releasing(i); SC(&OP[i].state, Endf);
    return (Succ, nil);
end.

UNLOCKING(i)/RELEASING(i)
begin
    for j := OP[i].N downto 1 do
        e.addr = OP[i].addr[j]; e.exp = OP[i].exp[j];
    again :
        x := LL(e.addr);
        if not VL(&OP[i].state) then return;
        if (x = (e.exp, nil)) or (x = (e.exp, i)) then
          if (not SC(e.addr, (e.exp, nil)) then
            goto again;
    return;
end.

UPDATING(i)
begin
    for j := 1 to OP[i].N do
        e.addr = OP[i].addr[j]; e.exp = OP[i].exp[j];
        e.new = OP[i].new[j];
    again :
        x := LL(e.addr]);
        if (not VL(&OP[i].state)) then return;
          if (x = (e.exp, i)) then
            if ( not SC(e.addr, (e.new, nil)) then goto again;
    return;
end.

LOCKING(helping, i, pos)
begin
start:
    for j := pos to OP[i].N do
        e.addr = OP[i].addr[j]; e.exp = OP[i].exp[j];
    again:
        x := LL(e.addr);
        if (not VL(&OP[i].state)) then return (Undecided, nil);
        if (x.value ≠ e.exp) then return (Fail, nil);
        else if (x.owner = nil) then
          if (not SC(e.addr, (e.exp, i)) then goto again;
        else if (x.owner ≠ i) then
          CheckingR(i, OP[i].blocked, j − 1);
          if (x.owner in casn_l) then return (CB, x.owner);
          Find index k in OP[x.owner] corresponding to x;
          ver = Version[x.owner];
          if ( not VL(e.addr)) then goto again;
          FAA(&OP[x.owner].blocked[k], 1);
          l := l + 1; casn_l[l] := x.owner;
          r := Help(helping, x.owner, k, ver);
          casn_l[l] := 0; l := l − 1;
          if ((r.kind = CB) and (r.value ≠ i)) then return r;
          goto start;
    return (Succ, nil);
end.

CHECKINGR(owner, blocked, kept)
begin
    if ((kept = 0) or (blocked = 0)) then return;
    if (not VL(&OP[owner].state)) then return;
    if ( blocked/kept > R* ) then SC(&OP[owner].state, Unlock);
    return;
end.
```

Figure 3. Procedures CASN, Help, Unlocking/Releasing, Updating, Locking and CheckingR in our first reactive multi-word compare-and-swap algorithm

- If it discovers a word having a value different from its old value passed to the CASN, its state changes into $Failure$ (line 8 in $Help$) and it starts phase two in order to release all the words it locked (line 11 in Help).
- If the unlock-condition is satisfied, its state changes to $Unlock$ (line 3 in $CheckingR$) and it starts to release the words it locked (line 3 in $Help$).

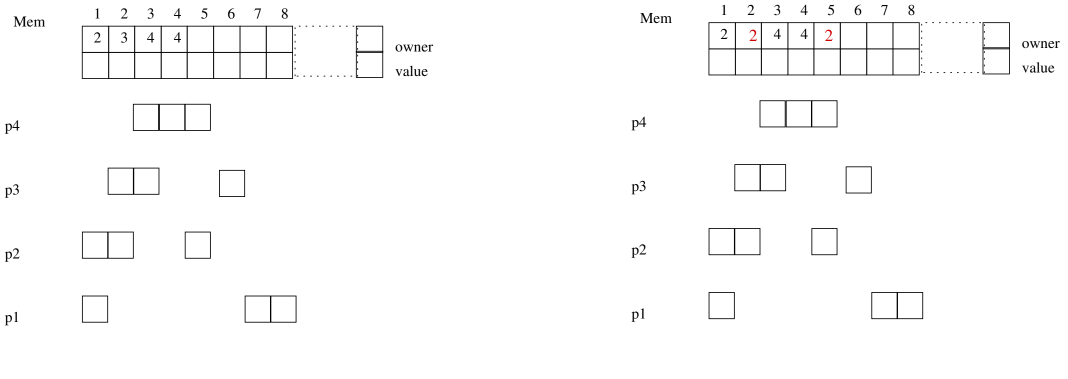
Procedure $Locking$ is the main procedure in phase one, which contains our first reactive scheme. In this procedure, the process called $helping$ tries to lock all N necessary words for $CASN_i$. If one of them has a value different from its expected value, the procedure returns $Fail$ (line 4 in $Locking$). It is worth noticing that the procedure returns $Fail$ regardless of whether the current value of the word is the new value that needs to be written to the word. The reason is that in the state $Lock$, the $CASN_i$ only can modify the field $owner$, not the field $value$, of the word. Otherwise, if the value of the word is the same as the expected value and it is locked by another CASN (lines 6-15 in $Locking$) and at the same time $CASN_i$ satisfies the unlock-condition, its state changes into $Unlock$ (line 3 in $CheckingR$). That means other processes whose CASNs blocked on the words acquired by $CASN_i$ can, on behalf of $CASN_i$, unlock the words and then acquire them while the $CASN_i$'s process helps its blocking CASN, $CASN_j$ (line 13 in $Locking$).

Procedure $CheckingR$ checks whether the average contention on the words acquired by $CASN_i$ is high and has passed a threshold: the unlock-condition. In this implementation, the *contention threshold* is R^*, $R^* = \sqrt{\frac{P-2}{N-1}}$, where P is the number of concurrent processes and N is the number of words needing to be updated atomically by CASN. The selection of this parameter is discussed in the full paper and is done in a similar way as in [4].

At time t, $CASN_i$ has created average contention r_i on the words that it has acquired, $r_i = \frac{blocked_i}{kept_i}$, where $blocked_i$ is the number of processes currently blocked by $CASN_i$ and $kept_i$ is the number of words currently locked by $CASN_i$. Because $CASN_i$ only checks the unlock-condition when: i) it is blocked and ii) it has locked at least a word and blocked at least a process (line 1 in $CheckingR$), we have $1 \leq blocked_i \leq (P-2)$ and $1 \leq kept_i \leq (N-1)$. The unlock-condition is to check whether $\frac{blocked_i}{kept_i} \geq R^*$. For each $CASN_i$, the variable $OP[i].blocked$ consists of N elements corresponding to the N words that need to be updated atomically. Every process blocked by $CASN_i$ on word $OP[i].addr[j]$ increases $OP[i].blocked[j]$ by one before helping $CASN_i$ using a fetch-and-add operation (FAA) (line 11 in $Locking$), and decreases the variable by one when it returns from helping the $CASN_i$ (line 5 and 9 in $Help$). The variable is not updated when the state of the $CASN_i$ is $Success$ or $Failure$ because in those cases $CASN_i$ no longer needs to check the unlock-condition.

There are two important variables in our algorithm, the $Version$ and *casn_l* variables. These variables are defined in Figure 3. The former is used for garbage collection purposes. That is when a process completes a CASN operation, the memory containing the CASN data, for instance $OP[i]$, can be used by a new CASN operation. Any process that wants to use $OP[i]$ for a new CASN must firstly increase the $Version[i]$ and pass the version to procedure Help (line 3 in $Casn$). When a process decides to help its blocking CASN, it must identify the current version of the CASN (line 9 and 10 in $Locking$) to pass to procedure $Help$ (line 13 in $Locking$). Assume process p_i is blocked by $CASN_j$ on word $e.addr$, and p_i decides to help $CASN_j$. If the version p_i reads at line 9 is not the version of $OP[j]$ at the time when $CASN_j$ blocked p_i, that is $CASN_j$ has ended and $OP[j]$ is re-used for another new CASN, the field *owner* of the word has changed. Thus, command $VL(e.addr)$ at line 10 returns failure and p_i must read the word again. This ensures that the version passed to $Help$ at line 13 in procedure $Locking$ is the version of $OP[j]$ at the time when $CASN_j$ blocked p_i. Then, before helping a CASN, processes always check whether the CASN version has changed (line 2 in $Help$).

The other significant variable is *casn_l*, which is local to each process and is used to trace which CASNs have been helped by the process in order to avoid the circle-helping problem. Consider the scenario described in Figure 4. Four processes p_1, p_2, p_3 and p_4 are executing four CAS3 operations: $CAS3_1$, $CAS3_2$, $CAS3_3$ and $CAS3_4$, respectively. The $CAS3_i$ is the CAS3 that is initiated by process p_i. At that time, $CAS3_2$ acquired $Mem[1]$, $CAS3_3$ acquired $Mem[2]$ and $CAS3_4$ acquired $Mem[3]$ and $Mem[4]$ by writing their original helping process identities in the respective owner fields (Recall that Mem is the set of *separate* words in the shared memory, not an array). Because p_2 is blocked by $CAS3_3$ and $CAS3_3$ is blocked by $CAS3_4$, p_2 helps $CAS3_3$ and then continues to help $CAS3_4$. Assume that while p_2 is helping $CAS3_4$, another process discovers that $CAS3_3$ satisfies the unlock-condition and releases $Mem[2]$, which was blocked by $CAS3_3$; p_1, which is blocked by $CAS3_2$, helps $CAS3_2$ acquire this $Mem[2]$ and then acquire $Mem[5]$. Now, p_2, when helping $CAS3_4$ lock $Mem[5]$, realizes that the word was blocked by $CAS3_2$, its own CAS3, that it has to help now. Process p_2 has made a cycle while trying to help other CAS3 operations. In this case, p_2 should return from helping $CAS3_4$ and $CAS3_3$ to help its own CAS3, because, at this time, the $CAS3_2$ is not blocked by any other CAS3. The local arrays *casn_l*s are used for this purpose. Each process p_i has a local array *casn_l*$_i$ with size of the maximal number of CASNs the process can help at one time. Recall that at one time each process can execute only one CASN, so the number is not greater than P, the number of processes in the system. In our implementation, we set the size of arrays $casn\_l$ to P, i.e. we do not limit the number of CASNs each process can help.



Before helping

After p_1 helps $CAS3_2$ acquire $Mem[2]$ and $Mem[5]$

Figure 4. Circle-helping problem

An element $casn\_l_i[l]$ is set to j when process p_i starts to help a $CASN_j$ initiated by process p_j and the $CASN_j$ is the l^{th} CASN that process p_i is helping at that time. The element is reset to 0 when process p_i completes helping the l^{th} CASN. Process p_i will realize the circle-helping problem if the identity of the CASN that process p_i intends to help has been recorded in $casn\_l_i$.

In procedure $Locking$, before process $p_{helping}$ helps $CASN_{x.owner}$, it checks whether it is helping the CASN currently (line 7 in $Locking$). If yes, it returns from helping other CASNs until reaching the unfinished helping task on $CASN_{x.owner}$ (line 15 in $Locking$). The l^{th} element of the array is set to $x.owner$ before the process helps $CASN_{x.owner}$, its l^{th} CASN, (line 12 in $Locking$) and is reset to zero after the process completes the help (line 14 in $Locking$).

In our methods, a process helps another CASN, for instance $CASN_i$, *just enough* so that its own CASN can progress. The strategy is illustrated by using the variable $casn\_l$ above and helping the $CASN_i$ unlock its words. After helping the $CASN_i$ release all its words, the process returns immediately because at this time the CASN blocked by $CASN_i$ can go ahead (line 5' in $Help$). After that, no process helps $CASN_i$ until the process initiating the CASN, p_i, returns and helps it progress (line 4 in $Help$).

3.2 Second Reactive Scheme

In the first reactive scheme, a $CASN_i$ must release all its acquired words when it is blocked and the average contention on these words is higher than a threshold, R^*. It will be more flexible if the $CASN_i$ can release only some of its acquired words on which many other processes are being blocked.

The second reactive scheme is more adaptable to contention variations on the shared data than the first one. An interesting feature of this method is that when process p_i is blocked, it only releases *just enough* words to reduce most of processes blocked by itself. Recall that r_i is the ratio of the number of processes blocked by $CASN_i$ to the number of words acquired by $CASN_i$. Therefore, when $CASN_i$ releases words with contention smaller than r_i, the average contention at that time, the next average contention will increase and $CASN_i$ must continue releasing words in decreasing order of word-indices until the word that made the average contention increase, is released. When this word is released, the average contention on the words locked by $CASN_i$ is going to reduce, and thus according to the first of the following rules, $CASN_i$ does not release its remaining words anymore at this time. That is how *just enough* to help most of the blocked processes is defined in our setting. The scheme is influenced by the idea of the *thread-based algorithm* [4].

In this second reactive scheme, the following rules must be satisfied in a transaction:

1. Release words only when the current contention is the highest so far.
2. When releasing, release *just enough* words to keep the competitive ratio $c = \varphi - \frac{\varphi-1}{\varphi^{1/(\varphi-1)}}$, where $\varphi = (P-2)*(N-1)$ for N-word CAS operations and P concurrent processes.

The number of words which should be released by a blocked $CASN_i$ at time t is $d_i^t = D_i * \frac{1}{c} * \frac{r_i^t - r_i^{t-1}}{r_i^t - \frac{1}{N-1}}$, where r_i^{t-1} is the highest contention until time $(t-1)$ and D_i is the number of words acquired by the $CASN_i$ at the beginning of the transaction. In our algorithm, r_i represents for the average contention on the words kept by a $CASN_i$ and is calculated by the following formula: $r_i = \frac{blocked_i}{kept_i}$ as mentioned in Section 3.1.

```
type state_type = record init; r_max; ul_pos; state; end;

HELP(helping, i, pos)
begin
start :
1:    gs := LL(&OP[i].state);
2:    if (ver ≠ Version[i]) then return (Fail, nil);
      if (gs.state = Unlock) then
4:       Unlocking(i, gs.ul_pos);
5:       cr = CheckingR(i, OP[i].blocked, gs.ul_pos, gs);
6:       if (cr = Succ) then goto start;
7:       else SC(&OP[i].state, Lock);
8:       if (helping = i) then goto start;
9:       else FAA(&OP[i].blocked[pos], −1); return (Succ, nil);
      else if (state = Lock) then
10:      result := Locking(helping, i, pos);
         if (result.kind = Succ) then
11:         SC(&OP[i].state, (0, 0, 0, Succ));
         else if (result = Fail) then
12:         SC(&OP[i].state, (0, 0, 0, Fail));
         else if (result.kind = CB) then
13:         FAA(&OP[i].blocked[pos], −1); return result;
         goto start;
      ...
end.

valtype READ(x)
begin
      y := LL(x);
      while (y.owner ≠ nil) do
            Help(self, y.owner); y := LL(x);
      return (y.value);
end.

UNLOCKING(i, ul_pos)
begin
      for j := OP[i].N downto ul_pos do
         e.addr = OP[i].addr[j]; e.exp = OP[i].exp[j];
      again :
         x := LL(e.addr);
         if (not VL(&OP[i].state)) then return;
         if (x = (e.exp, nil)) or (x = (e.exp, i)) then
            if (not SC(e.addr, (e.exp, nil)) then goto again;
      return;
end.

CHECKINGR(owner, blocked, kept, gs)
begin
      if (kept = 0) or (blocked = 0)) then return Fail;
1:    if ( not VL(&OP[owner].state)) then return Fail;
      for j := 1 to kept do nb := nb + blocked[j];
1':   r := nb/kept;
      if (r < m * C) then return Fail; /* m = 1/(N−1) */
2:    if (kept ≠ gs.ul_pos) then
         d = kept * 1/C * (r−m*C)/(r−m); ul_pos := kept − d + 1;
3:       SC(&OP[owner].state, (kept, r, ul_pos, Unlock));
         return Succ;
4:    else if (r > gs.r_max) then
         d = gs.init * 1/C * (r−gs.r_max)/(r−m); ul_pos := kept − d + 1;
5:       SC(&OP[owner].state, (gs.init, r, ul_pos, Unlock));
         return Succ;
      return Fail;
end.
```

Figure 5. Procedures Help, Unlocking, CheckingR in our second reactive multi-word compare-and-swap algorithm and the procedure for Read operation

According to rule 1, contention r_i is considered for adjustment only if it increases, i.e. when either the number of processes blocked on the words kept by $CASN_i$ increases or the number of words kept by $CASN_i$ decreases. Therefore, in this implementation, which is described in Figure 5, the procedure $CheckingR$ is called not only from inside the procedure $Locking$ as in the first algorithm, but also from inside the procedure $Help$ when the number of words locked by $CASN_i$ reduces (line 5). In our algorithm, the variable $OP[i].blocked[j]$ is updated in such a way that the number of processes blocked on each word is known, and thus a process helping $CASN_i$ can calculate how many words need to be released and release *just enough* words. To be able to perform this task, besides the information about contention r_i, which is calculated through variables $blocked_i$ and $kept_i$, the information about the highest r_i so far and the number of words locked by $CASN_i$ at the beginning of the transaction is needed. This additional information is saved in two new fields of $OP[i].state$ called $init$ and r_{max}. While the $init$ is updated only one time at the beginning of the transaction (line 3 in $CheckingR$), the r_{max} field is updated whenever the unlock-condition is satisfied (line 3 and 5 in $CheckingR$). After calculating the number of words to be released, the position from which the words are released is saved in field ul_pos of $OP[i].state$ and called ul_pos_i. Consequently, the process helping $CASN_i$ will only release the words from $OP[i].addr[ul\_pos_i]$ to $OP[i].addr[N]$ through the procedure $Unlocking$. If $CASN_i$ satisfies the unlock-condition even after a process has just helped it unlock its words, the same process will continue helping the $CASN_i$ (line 5 and 6 in $Help$). Otherwise, if the process is not process p_i, the process initiating $CASN_i$, it will return to help the CASN that was blocked by $CASN_i$ before (line 9 in $Help$). The changed procedures compared with the first implementation are $Help$, $Unlocking$ and $CheckingR$, which are described in Figure 5.

4 Correctness Proof

In this section, we prove the correctness of our methods. The Figure 6 briefly describes shared variables and procedures reading or directly updating them.

	Mem	OP[i].state	OP[i].blocked
Help(helping, i, pos, ver)		LL, *SC*	*FAA*
Unlocking(i, ul_point)	LL, *SC*	VL	
Releasing(i)	LL, *SC*	VL	
Updating(i)	LL, *SC*	VL	
Locking(helping, i, pos)	LL, *SC*	VL	*FAA*
CheckingR(owner, blocked, kept, gs)		VL, *SC*	
Read(x)	LL		

Figure 6. Shared variables with procedures reading or directly updating them

The array OP consists of P elements, each of which is updated by only one process, for example $OP[i]$ is only updated by process p_i. Without loss of generality, we only consider an element of array OP, $OP[i]$, on which many concurrent helping processes get necessary information to help a CASN, $CASN_i^j$. Symbol $CASN_i^j$ means that this CASN uses variable $OP[i]$ and that it is the j^{th} time the variable is re-used. The value of $OP[i]$ read by process p_k is *correct* if it is the data of the CASN that blocks the CASN helped by p_k. For example, p_k helps $CASN_{i1}^{j1}$ and realizes the CASN is blocked by $CASN_i^j$. Thus, p_k decides to help $CASN_i^j$. But if the value p_k read from $OP[i]$ is the value of the next CASN, $CASN_i^{j+1}$, i.e. $CASN_i^j$ has completed and $OP[i]$ is re-used for another new CASN, the value is not correct to p_k.

Lemma 1: Every helping process reads correct values of variable $OP[i]$.

Proof. In our pseudo-code described in Figure 3 and Figure 5, the value of $OP[i]$ is read before the process checks $VL(\&OP[i].state)$ (line 1 in $Unlocking$, $Releasing$, $Updating$ and $Locking$). If $OP[i]$ is re-used for $CASN_i^{j+1}$, the value of $OP[i].state$ has certainly changed since p_k read it at line 1 in procedure $Help$ because $OP[i]$ is re-used only if the state of $CASN_i^j$ has changed into $Ends$ or $Endf$. In this case, p_k realizes the change and returns to procedure $Help$ to read the new value of $OP[i].state$ (line 1 in $Help$). In procedure $Help$, p_k will realize that $OP[i]$ is reused by checking its version (line 2 in $Help$) and return, that is p_k do not use incorrect values to help CASNs. Moreover, when p_k decides to help $CASN_i^j$ at line 13 in procedure $Locking$, the version of $OP[i]$ passed to the procedure $Help$ is correct version, that is the version corresponding to $CASN_i^j$, the CASN blocking the current CASN on word $e.addr$. If the version p_k read at line 9 in procedure $Locking$ is incorrect, that is $CASN_i^j$ has completed and $OP[i]$ is re-used for $CASN_i^{j+1}$, p_k will realize this by checking $VL(e.addr)$. Because if $CASN_i^j$ has completed, the $owner$ field of word $e.addr$ will change from i to nil. Therefore, p_k will read the value of word $e.addr$ again and realize $OP[i].state$ has changed. In this case, p_k will return and not use the incorrect data as argued above. □

From this point, we can assume that the value of $OP[i]$ used by processes is correct. In our reactive compare-and-swap (RCASN) operation, the linearization point is the point its state changes into $Succ$ if it is successful or the point when its state changes into $Fail$ if it fails. It is easy to realize that our specific $Read$ operation[2] is linearizable to RCASN. The $Read$ operation is the same as those in [8][12]. Thus, what we need to prove is that the interferences among the procedures do not cause anything wrong.

Lemma 2: The interferences among the procedures do not affect on the correct results.

Proof. We focus on the changes of $OP[i].state$. Assume the latest change occurs at time t_0. The processes helping $CASN_i$ are divided into two group: group two consists of the processes realizing the change and the rest is in group one. The correct results are made by the processes in group two. What we need to prove is that:

1. The processes in group one do not affect on the correct results made by the processes in group two,

[2] The $Read$ procedure in figure 5 is used for both reactive CASN algorithms

2. The interferences between processes in group two do not cause unexpected results.

To prove the first statement, we consider all cases where a process in group two can be interfered by processes in group one. Let p_l^j is a process p_l in group j.

Case 1.1 : Assume p_k^1 interferes p_l^2 while p_l^2 is executing procedure $Help$ or $CheckingR$. Because the procedures only use shared variable $OP[i].state$, to interfere p_l^2 the procedure p_k^1 is executing must update the variable, i.e. p_k^1 must be executing procedure $Help$ or $CheckingR$. However, because p_k^1 do not realize the change of $OP[i].state$, that is the change occurs after p_k^1 read $OP[i].state$ by LL at line 1 in $Help$, it will fail when updating $OP[i].state$ by SC. Therefore, p_k^1 can't interfere p_l^2 while p_l^2 is executing procedure $Help$ or $CheckingR$, or in other words, p_l^2 can't be interfered through shared variable $OP[i].state$.

Case 1.2 : p_l^2 is interfered through a shared variable $Mem[x]$ while executing procedures $Unlocking$, $Releasing$, $Updating$ or $Locking$. Because $OP[i].state$ changed after p_k^1 read it and in the new state of $CASN_i$ p_l^2 must update $Mem[x]$, the state p_k^1 read is only able to be $Lock$ or $Unlock$. Thus, the value p_k^1 can write to $Mem[x]$ is $(OP[i].exp[y], i)$ or $(OP[i].exp[y], nil)$, where $OP[i].addr[y]$ points to $Mem[x]$, and only if $Mem[x]$ is $(OP[i].exp[y], nil)$ or $(OP[i].exp[y], i)$. The followings are all cases where p_k^1 is possible to modify $Mem[x]$, so that it can interfere p_l^2.

Case 1.2.1 : The current value of $Mem[x]$ is $(OP[i].exp[y], i)$ and p_k^1 is executing $Unlocking$ to update it to $(OP[i].exp[y], nil)$, i.e the state of $CASN_i$ p_k^1 read is $Unlock$. Because the state of $CASN_i$ only can changes from $Unlock$ to $Lock$ and the change occurs only if every element of Mem, for instance $Mem[x]$, needing to be unlocked has updated to $(OP[i].exp[y], nil)$, the current value of $Mem[x]$ is $(OP[i].exp[y], i)$ only if a process in the next state, $Lock$, has modified the variable. Therefore, p_k^1 will fail to modify the variable (line 4 in $Unlocking$). In this case, p_k^1 can not interfere p_l^2.

Case 1.2.2 : The current value of $Mem[x]$ is $(OP[i].exp[y], nil)$. In this case, p_k^1 can modify the variable through procedure $Unlocking$ or $Locking$ and p_l^2 can be executing $Unlocking$, $Releasing$ or $Locking$ (procedure $Updating$ is only called when all necessary words have been locked by $CASN_i$).

If p_k^1 modifies $Mem[x]$ successfully (line 4 in $Unlocking$ or line 5 in $Locking$), p_l^2 will realize the modification and read $Mem[x]$ again (line 4 in $Unlocking$/$Releasing$ and line 5 in $Locking$). If the value after the modification is still $(OP[i].exp[y], nil)$ or is $(OP[i].exp[y], i)$ and p_l^2 is executing $Unlocking$ or $Releasing$, p_l^2 will overwrite the correct value on the variable. If p_l^2 is executing $Locking$, p_k^1 can not successfully write $(OP[i].exp[y], i)$ to $Mem[x]$ through $Locking$ because the state of $OP[i].state$ can not directly change from $Lock$ to $Lock$, but it must pass the intermediate state $Unlock$. Thus, at the time when the state of $CASN_i$ changes from $Lock$ to $Unlock$, even if p_k^1 modified $Mem[x]$ successfully, the variable would be overwritten by the process executing procedure $Unlocking$.

If p_k^1 modifies $Mem[x]$ after p_l^2 executed a procedure such as $Unlocking$, $Releasing$ or $Locking$ on the variable $Mem[x]$, p_k^1 will fail because all these procedures update $Mem[x]$ when its value is $(OP[i].exp[y], nil)$ (line 4 in $Unlocking$/$Releasing$ and line 5 in $Locking$). Therefore, in this case p_k^1 can not interfere p_l^2

From case 1.2.1 and 1.2.2, it conduces that p_k^1 can not interfere p_l^2 through the shared variable $Mem[x]$.

In conclusion, the processes in group one can not interfere the processes in group two via the shared variables, that is the first statement is proved.

Lastly, we prove the second statement: the interferences among the processes in group two do not cause unexpected results. On the shared variable $OP[i].state$, only the processes executing procedure $Help$ or $CheckingR$ can interfere with one another. In this case, the linearization point is when the processes modify the variable by SC. On the shared variable $Mem[x]$, all processes in group two will execute the same procedure such as $Unlocking$, $Releasing$, $Updating$ and $Locking$, because they read the latest state of $OP[i].state$ Therefore, the procedures are executed as if they are executed by one process without any interference. In conclusion, the interferences among the processes in group two do not cause any unexpected results. □

Lemma 3: A $CASN_i$ blocks another $CASN_j$ at only one position in Mem for all times $CASN_j$ is blocked by $CASN_i$.

Proof. We prove *Lemma 3* by contradiction. Assume $CASN_j$ is blocked by $CASN_i$ at two position x_1 and x_2 on the shared variable Mem, where $x_1 < x_2$. At the time $CASN_j$ is blocked at x_2, both $CASN_i$ and $CASN_j$ must have acquired x_1 already because the CASN tries to acquire an item on Mem only if all the lower items it needs have been acquired by itself. This is a contradiction because an item is only acquired by one CASN. □

The following lemmas prove that our methods satisfy the requirements of online-search and one-way trading algorithms [4].

Lemma 4: Whenever the average contention on acquired words *increases during a transaction, the unlock-condition is checked.*

Proof. According to our algorithm, in procedure $Locking$, every time a process increases $OP[owner].blocked$, it will help $CASN_{owner}$. If the CASN is in a transaction, i.e. being blocked by another CASN, for instance $CASN_j$, the process will certainly call $CheckingR$ to check unlock-condition. Additionally, in our second method the average contention can increase when the CASN releases some of its words and the increment is checked at line 5 in procedure $Help$. □

Lemma 4 has the important consequence that the process always get the *average contention on the acquired words* of a CASN whenever it increases, so applying online-search and one-way trading algorithms with the value the process obtains is correct according to the algorithm.

Lemma 5: Procedure $CheckingR$ checks the unlock-condition correctly.

Proof. Assume that process p_m executes $CASN_i$ and then realizes that $CASN_i$ is blocked by $CASN_j$ on word $OP[i].addr[x]$ at time t_0 and read $OP[i].blocked$ at time t_1. Between t_0 and t_1 the other processes which are blocked by $CASN_i$ can update $OP[i].blocked$. Because $CheckingR$ only sums on $OP[i].blocked[k]$, where $k = 1, .., x - 1$, only processes blocked on words from $OP[i].addr[1]$ to $OP[i].addr[x - 1]$ are counted in $CheckingR$. These processes updating $OP[i].blocked$ is completely independent of the time when $CASN_i$ was blocked on word $OP[i].addr[x]$. Therefore, this situation is similar to one where all the updates happen before t_0, i.e. the value of $OP[i].blocked$ used by $CheckingR$ is the same as one in a sequential execution without any interference between the two events that $CASN_i$ is blocked and that $OP[i].blocked$ is read. Therefore, the unlock-condition is checked correctly. Moreover, if $CASN_i$'s state is changed to Unlock, the words from $OP[i].addr[x]$ to $OP[i].addr[N]$ acquired by $CASN_i$ after time t_0 due to another process's help, will be also released. This is the same as a sequential execution: if $CASN_i$'s state is changed to Unlock at time t_0, no further word can be acquired. □

5 Evaluation

We compared our algorithms to the two best previously known alternatives: i) the algorithm presented in [12] that is the best representative of the *recursive helping* policy (RHP), and ii) the algorithm presented in [14] that is an improved version of the *software transactional memory* [16] (iSTM). In the latter, a dummy function that always returns zero is passed to CASN. Regarding the multi-word compare-and-swap operation in [8], the lock-free memory management scheme in this algorithm is not clearly described. When we tried to implement it, we did not find any way to do so without facing live-lock scenarios or using blocking memory management schemes. Their implementation is expected to be released in the future [9], but is not currently available. However, relying on the experimental data of the paper [8], we can conclude that this algorithm performs approximately as fast as iSTM in the shared memory size range from 256 to 4096 with sixteen threads.

The system used for our experiments was a ccNUMA SGI Origin2000 with thirty two 250MHz MIPS R10000 CPUs with 4MB L2 cache rated at 14.7 SPECint95 and 24.5 SPECfp95 each. The system ran IRIX 6.5 and it was used exclusively. An extra processor was dedicated for monitoring. The Load-Linked (LL), Validate (VL) and Store-Conditional (SC) instructions used in these implementations were implemented from the LL/SC instructions supported by the MIPS hardware [13]. The shared memory Mem is divided into N parts, and the i^{th} word in N words needing to be updated atomically is chosen randomly in the i^{th}part of the shared memory to ensure that words pointed by $OP[i].addr[1]$ to $OP[i].addr[N]$ are in the inscreasing order of their indices on Mem. The paddings are inserted between every pair of adjacent words in Mem to put them on seperate cache lines. The values that will be written to words of Mem are contained in two-dimension array $Value[3][N]$. The value of $Mem[i]$ will be updated to $Value[1][i]$, $Value[2][i]$, $Value[3][i]$, $Value[1][i]$, and so on, so that we do not need to use procedure Read, which also uses procedure Help, to get the current value of $Mem[i]$. Therefore, the time in which only the CASN operations are executed is measured more accurately. The CPU time is the average of the useful time on each process, the time only used for CASNs. The useful time is calculated by subtracting total time by overhead time. The number of successful CASNs is the sum of the numbers of successful CASNs on each process.

In each experiment, all CASN operations concurrently ran on thirty processors for one minute. The time spent on CASN operations was measured. The contention on the shared memory Mem was controlled by its size. When the size of shared memory was 32, running eight-word compare-and-swap operations caused a high contention environment. When the size of shared memory was 16384, running two-word compare-and-swap operations created a low contention environment because the probability that two CAS2 operations competed for the same words was small.

5.1 Results

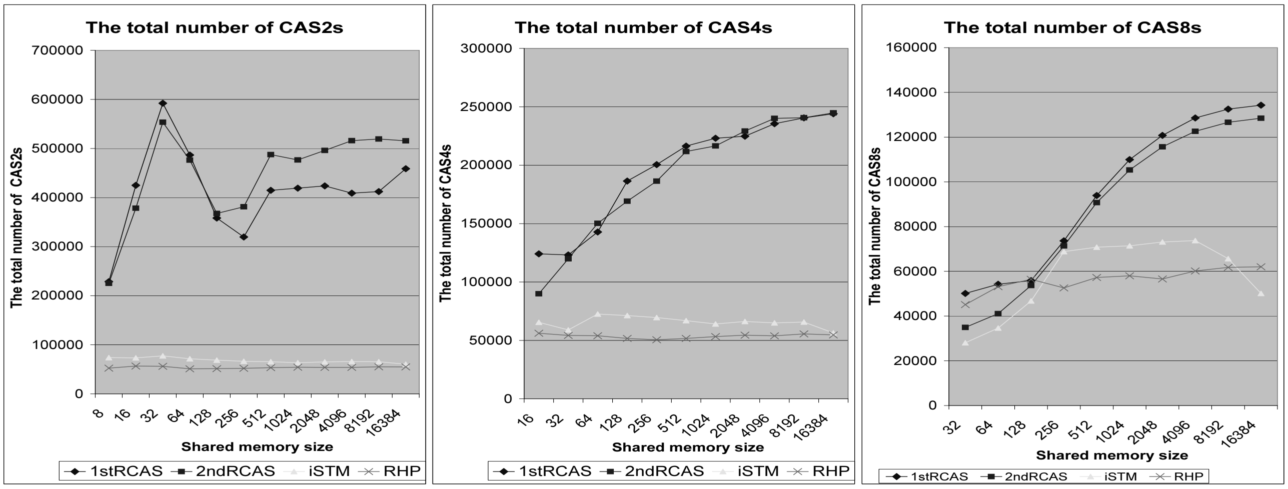


Figure 7. The number of CAS2s, CAS4s and CAS8s in one second

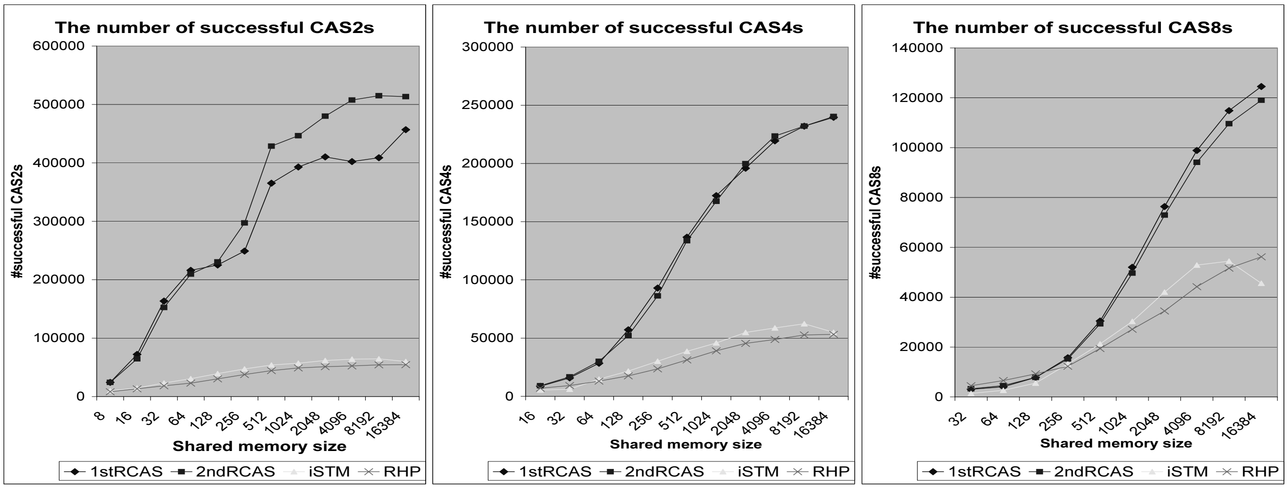


Figure 8. The number of *successful* CAS2s, CAS4s and CAS8s in one second

The results show that our CASN constructions compared to the previous constructions are significantly faster for almost all cases. Figure 7 describes the number of CASN operations performed in one second by the different constructions.

In order to analyse the improvements that are because of the reactive behaviour, let us first look at the results for the extreme case where there is almost no contention and the reactive part is rarely used: CAS2 and the shared memory size of 16384. In this extreme case, only the efficient design of our algorithms gives the better performance. In the other extreme case, when the contention is high, for instance the case of CAS8 and the shared memory size of 32, the brute force approach of the recursive helping scheme (RHP) is the best strategy to use. When there are too many conflicts between different CASN operations, the best way is to go sequentially and help the processes recursively. Our reactive schemes start helping the performance of our algorithms when the contention coming from conflicting CASN operations is not at its full peak. In these cases, the decision on whether to release the acquired words plays the role in gaining performance. The benefits from the reactive schemes come quite early and drive the performance of our algorithms to reach their best performance rapidly. Figure 7 shows that the chart of RHP is nearly flat regardless of the contention whereas those of our reactive schemes increase rapidly with the decrease of the contention.

Figure 8 describes the number of *successful* CASN operations performed in one second by the different constructions. The results are similar in nature with the results described in the previous paragraph. In full contention, the recursive scheme works quite well because in high contention the conflicts between different CASN operations can not be really solved locally by each operation and thus the serialized version of the recursive help is the best that we can hope for. However, when the contention is not at its full peak, our reactive schemes catch up fast and help the processes to solve their conflicts locally.

Both figures show that our algorithms outperform the best previous alternatives in almost all cases. At the memory size 16384 in Figure 7, our first reactive two-word compare-and-swap (1stRCAS) is more than seven times faster than both RHP and iSTM, and our second one (2ndRCAS) is about nine times faster than both RHP and iSTM. On the four-word compare-and-swap, both our RCAS are more than four time faster than both RHP and iSTM. On the eight-word compare-and-swap, both our RCAS are more than two time faster than both RHP and iSTM.

Regarding the number of successful CASN operations, our RCAS still outperform RHP and iSTM in almost all cases. At the memory size of 16384 in Figure 8, our 1stRCAS gives more than seven times as many successful CAS2s as both RHP and iSTM and our 2ndRCAS gives about nine times as many successful CAS2s as both RHP and iSTM. Both our RCAS give more than four times as many successful CAS4s as both RHP and iSTM and more than two times as many successful CAS8s as both RHP and iSTM.

6 Conclusion

Multi-word synchronization constructs are important for multiprocessor systems. Two reactive, lock-free algorithms that implement multi-word compare-and-swap operations are presented in this paper. The key to these algorithms is for every CASN operation to measure in an efficient way the contentions on the words it has acquired, and reactively decide whether and how many words need to be released. Our algorithms are also designed in an efficient way that allows high parallelism — both algorithms are lock-free— and most significantly, guarantees that the new operations spend significantly less time when accessing coordination shared variables usually accessed via expensive hardware operations. The algorithmic mechanism that measures contention and reacts accordingly is efficient and does not cancel the benefits in most cases. Our algorithms also promote the CASN operations that have higher probability of success among the CASNs generating the same contention. Experiments on thirty processors of a SGI Origin2000 multiprocessor show that both our algorithms react fast according to the contention conditions and significantly outperform the best-known alternatives in all contention conditions. Our algorithms are expected to be incorporated in the NOBLE synchronization library [17].

References

[1] J. H. Anderson and M. Moir. Universal Constructions for Multi-Object Operations. *Proceedings of the 14th Annual ACM Symposium on the Principles of Distributed Computing*, pp. 184–193, August 1995

[2] J. H. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects on priority-based systems. *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 229–238, August 1997.

[3] G. Barnes. A Method for Implementing Lock-Free Shared Data Structures. *ACM Symposium on Parallel Algorithms and Architectures*, pp. 261–270, 1993.

[4] R. El-Yaniv, A. Fiat, R. M. Karp, G. Turpin Optimal Search and One-Way Trading Online Algorithms. *Algorithmica* 30:101-139 (2001).

[5] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. *Operating Systems Design and Implementation*, pp. 123–136, 1996.

[6] M. Greenwald Non-blocking synchronization and system design. *PhD thesis*, Stanford University, August 1999. Technical report STAN-CS-TR-99-1624.

[7] M. Greenwald. Two-Handed Emulation: How to build Non-Blocking implementations of Complex Data-Structures using DCAS. *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pp. 260–269, July 2002.

[8] T. L. Harris and K. Fraser and I. A Pratt. A practical multi-word compare-and-swap operation. *16th International Symposium on Distributed Computing (DISC 2002)*, pp. 265-279

[9] T. L. Harris *Personal Communication*, August 2002.

[10] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745-770, November 1993.

[11] M. Herlihy. Wait-Free Synchronization. *ACM TOPLAS*, Vol. 11, No. 1, pp. 124-149, Jan. 1991.

[12] A. Israeli and L. Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pp. 151–160, August 1994.

[13] M. Moir. Practical Implementations of Non-Blocking Synchronization Primitives *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, pp. 219–228, August 1997

[14] M. Moir. Transparent support for wait-free transactions. *Distributed Algorithms, 11th International Workshop, volume 1320 of Lecture Notes in Computer Science*, pp. 305– 319. Springer-Verlag, September 1997.

[15] D. S. Nikolopoulos and T. S. Papatheodorou. The Architectural and Operating System Implications on the Performance of Synchronization on ccNUMA Multiprocessors. *International Journal of Parallel Programming* Volume 29, No. 3, June 2001

[16] N. Shavit and D. Touitou. Software Transactional Memory. *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pp. 204–213, August 1995.

[17] H. Sundell, P. Tsigas. NOBLE: A Non-Blocking Inter-Process Communication Library. *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers (LCR'02)*, Lecture Notes in Computer Science, Springer Verlag, 2002.

[18] P. Tsigas, Y. Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. *Proceedings of the international conference on Measurement and modelling of computer systems (SIGMETRICS 2001)*, pp. 320-321, ACM Press, 2001.

[19] P. TSIGAS, Y. ZHANG. Integrating Non-blocking Synchronisation in Parallel Applications: Performance Advantages and Methodologies. *Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP '02)*, ACM Press, 2002.

[20] The manpages of SGI Origin 2000.