Journal of Instruction-Level Parallelism 6 (2004) 1-25 Submitted 1/04; published 4/04

Reactive Multi-word Synchronization for Multiprocessors*

Phuong Hoai Ha PHUONG@CS.CHALMERS.SE
Philippas Tsigas TSIGAS@CS.CHALMERS.SE
Department of Computer Science
Chalmers University of Technology
S-412 96 Göteborg, Sweden

Abstract

Shared memory multiprocessor systems typically provide a set of hardware primitives in order to support synchronization. Generally, they provide *single-word* read-modify-write hardware primitives such as compare-and-swap, load-linked/store-conditional and fetch-and-op, from which the higher-level synchronization operations are then implemented in software. Although the *single-word* hardware primitives 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 as well as the memory conflicts of the *multi-word* compare-and-swap operations, and in response, they react accordingly in order to guarantee good performance in a wide range of system conditions. The algorithms are non-blocking (lock-free), allowing in this way fast dynamical behavior. Experiments on thirty processors of an SGI Origin2000 multiprocessor show that both our algorithms react quickly according to the contention variations and outperform the best known alternatives in almost all contention conditions.

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 hardware and how much 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, an *old* value and a *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, an N-word compare-and-swap operation takes the addresses, *old* values and *new* values of N words, and if the current contents of these N words all are the same as the respective *old* values, the CASN will update the new values to the respective words atomically. Otherwise, we say that the CAS/CASN fails, leaving the variable

*. This work was partially supported by the Swedish Research Council (VR).

©2004 AI Access Foundation and Morgan Kaufmann Publishers. All rights reserved.

values unchanged. 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. 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-Linked/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 may decide to 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, does not release the words it has acquired until its failure is definite, 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 significantly increase its execution time but also increase the contention in the system when it tries to acquire these words again. The disadvantage of these strategies is that both of them are not adaptable to the different memory access patterns that different CASNs can trigger, or to frequent variations of the contention on each individual 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 in most contention situations. Of course, dynamically changing the behavior 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 behavior 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. Section 3 presents our algorithms at an abstract level. The algorithms in detail are described in Section 4. Section 5 presents the correctness proofs of our algorithms. In Section 6 we present the performance evaluation of our CASN algorithms and compare them to the best known alternatives, which also represent the two helping strategies mentioned above. Finally, Section 7 concludes the paper.

2. Problem Description, Related Work and Our Contribution

Concurrent data structures play a significant role in multiprocessor 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 cannot 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. *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 the 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 interferences among processes are 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 the loss of disjoint-access-parallelism. The *disjoint-access-parallelism* means that processes accessing no common portion of the shared data should be able to progress in parallel.

Barnes [3] later suggested a *cooperative technique* which allows many processes to access the same data structure concurrently as long as the processes write down exactly what they will be doing. Before modifying a portion of the shared data, a process p_1 checks whether this portion is used by another process p_2. If this is the case, process p_1 will cooperate with p_2 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-parallelism was introduced. All processes try to lock all portions of the shared data they need before writing back the new values to the portions one by one. An *owner* field is assigned to every portion of the shared data to inform the processes about which process is the owner of the portion at that time.

Harris, Fraser and Pratt [8] aiming to reduce the per-word space overhead eliminated the owner field. They exploited the data word for containing a special value, a pointer to a CASNDescriptor, to pass the information of which process is the owner of the data word. However, in their paper the memory management problem is not discussed clearly.

A wait-free multi-word compare-and-swap was introduced by Anderson and Moir in [1]. The cooperative technique was employed in the aforementioned results as well.

However, the disadvantage of the *cooperative technique* 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 may be able to make

progress if these words are released. This *cooperative technique* uses a *recursive helping policy*, and the time needed 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.

The left part in Figure 1 illustrates the helping strategy of the *recursive helping policy*. There are three processes executing three CAS4: p_1 wants to lock words 1,2,3,4; p_2 wants to lock words 3,6 and two other words; and p_3 wants to lock words 6,7,8 and another word. At that time, the first CAS4 acquired words 1 and 2, the second CAS4 acquired word 3 and the third CAS4 acquired words 6,7 and 8. When process p_1 helps the first CAS4, it realizes that word 3 was acquired by the second CAS4 and thus it helps the second CAS4. Then, p_1 realizes that word 6 was acquired by the third CAS4 and it continues to help the third CAS4 and so on. We observe that i) the time for a process to help other CASN operations may be long and unpredictable and ii) if the second CAS4 did not conservatively keep word 3 while helping other CAS4, the first CAS4 could succeed without helping other CAS4s, especially the third CAS4 that did not block the first CAS4. Note that helping causes more contention on the memory. Therefore, the less helping is used, the lower the contention level on the memory is.

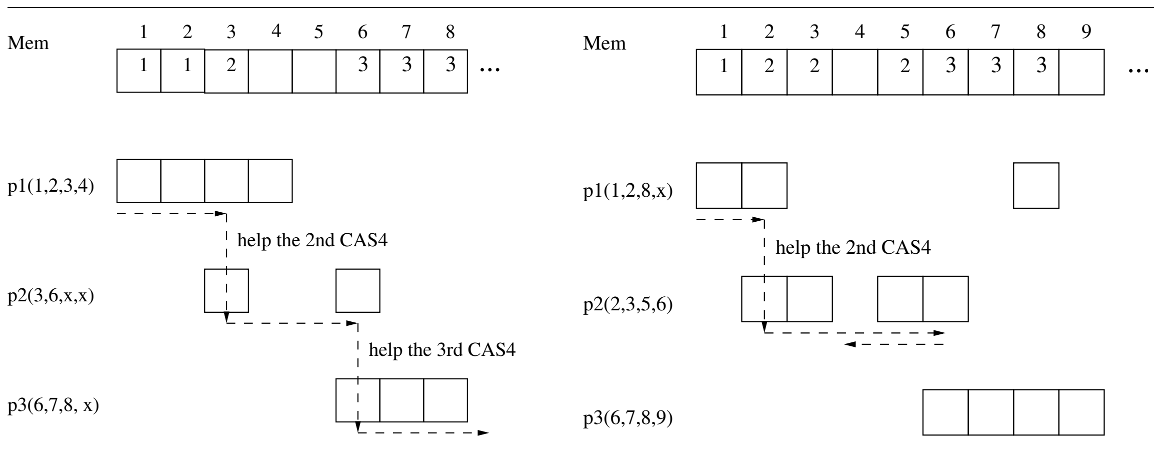

Figure 1: Recursive helping policy and software transactional memory

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 one blocking process, so *recursive helping* does not occur. STM then was improved by 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 contention level 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 may have to compete with other processes to acquire the words again. Moreover, even if a process acquired the whole set of words it needs except for the last one, which is owned by another process, it still releases all the words and then starts from scratch. In this case, it should realize that not many processes require the words and that it is almost successful, so it would be best to try to keep the words as in the *cooperative technique*.

The right part of Figure 1 illustrates the helping strategy of STM. At that time, the first CAS4 acquired word 1, the second CAS4 acquired words 2,3 and 5 and the third CAS4 acquired words 6,7 and 8. When process p_1 helps the first CAS4, it realizes that word 2 was acquired by the second CAS4. Thus, it releases word 1 and helps the second CAS4. Then, when p_1 realizes that the second CAS4 was blocked by the third CAS4 on word 6, it, on behalf of the second CAS4, releases word 5,3 and 2 and goes back to help the first CAS4. Note that i) p_1 could have benefited by keeping word 1 because no other CAS4 needed the word; otherwise, after helping other CAS4s, p_1 has to compete with other processes to acquire word 1 again; and ii) p_1 should have tried to help the second CAS4 a little bit more because this CAS4 operation was close to success.

Note that most algorithms require the N words to be sorted in addresses and this can add an overhead of $O(logN)$ because of sorting. However, most applications can sort these addresses before calling the CASN operations.

2.1 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 contention levels while *bigger* structures and more *complex* mechanisms can help to distribute processors/processes among the 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 favors the operations closer to completion. The key to our second algorithm is for a CASN to release not *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 of the proposed algorithms on thirty processors of an SGI Origin2000 multiprocessor, which is presented in Section 6, matches our intuition. In particular, it shows that both our algorithms react fast according to the contention conditions and significantly outperform the best-known alternatives in all contention conditions.

3. Algorithm Informal Description

In this section, we present the ideas of our reactive CASN operations at an abstract level. The details of our algorithms are presented in the next section.

In general, practical CASN operations are implemented by locking all the N words and then updating the value of each word one by one accordingly. Only the process having acquired all the N words it needs can try to write the new values to the words. The processes that are blocked, typically have to *help* the blocking processes so that the lock-free feature is obtained. The helping schemes presented in [1, 8, 12, 14, 16] are based on different strategies that are described in Section 2.

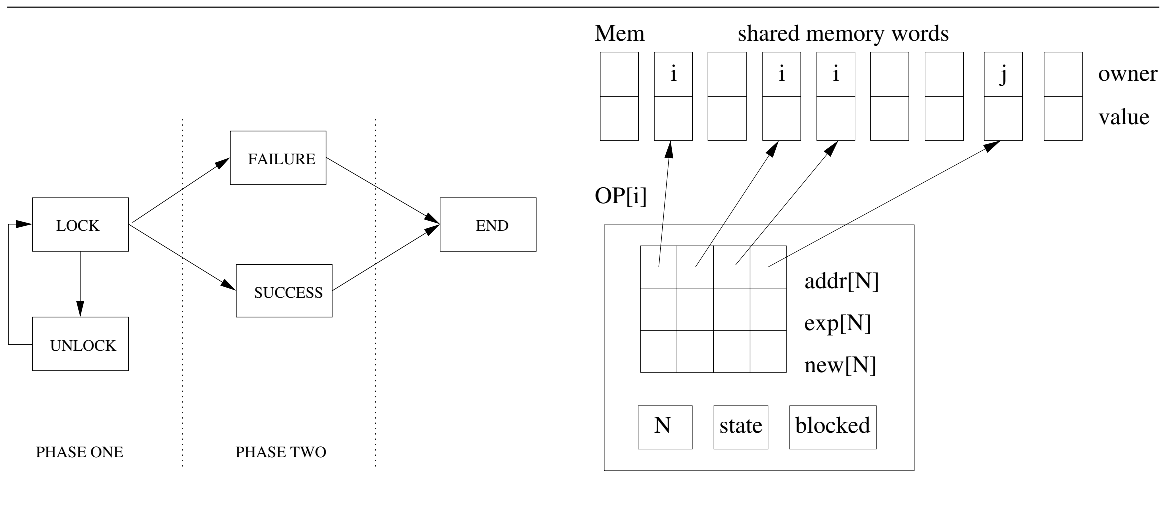

Figure 2: Reactive-CASN states and reactive-CAS4 data structure

The variable $OP[i]$ described in Figure 2 is the shared variable that carries the data of $CASN_i$. It consists of three arrays with N elements each: $addr_i$, exp_i and new_i and a variable $blocked_i$ 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 CASN operations blocked on the words, respectively. The N elements of array $addr_i$ must be increasingly sorted in addresses to avoid live-lock in the helping process. Each entry of the shared memory Mem, a normal 32-bit word used by the real application, has two fields: the *value* field (24 bits) and the *owner* field (8 bits). The *owner* field needs $log_2 P + 1$ bits, where P is the number of processes in the system. 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. For a system supporting 64-bit words, the value field can be up to 56 bits if $P < 256$. However, the value field cannot contain a pointer to a memory location in some modern machines where the size of pointer equals the size of the largest word. For information on how to eliminate the owner field, see [8].

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 according to the result of phase 1.

Figure 3 describes our CASN operation at a high level.

CASN($OP[i]$)
try_again :
  Try to lock all N necessary words;
  if *manage to lock all the N words* **then**
    write new values to these words one by one; **return** *Success*;
  else if *read an unexpected value* **then**
    release all the words that have been locked by $CASN_i$; **return** *Failure*;
  else if *contention is "high enough"* **then**
    release some/all $CASN_i$*'s words to reduce contention*; **goto** *try_again*;

Figure 3: Reactive CASN description

In order to know whether the contention on $CASN_i$'s words is high, each $CASN_i$ uses variable $OP[i].blocked$ to count how many other CASNs are being blocked on its words. Now, which contention levels should be considered *high*? The $CASN_i$ has paid a price (execution time) for the number of words that it has acquired and thus it should not yield these words to other CASNs too generously as the *software transactional memory* does. However, it should not keep these words egoistically as in the *cooperative technique* because that will make the whole system slowdown. Let w_i be the number of words currently kept by $CASN_i$ and n_i be the estimated number of CASNs that will go ahead if some of w_i words are released. The $CASN_i$ will consider releasing its words only if it is blocked by another CASN. The challenge for $CASN_i$ is to *balance the trade-off* between its own progress w_i and the potential progress n_i of the other processes. If $CASN_i$ knew how the contention on its w_i words will change in the future from the time $CASN_i$ is blocked to the time $CASN_i$ will be unblocked, as well as the time $CASN_i$ will be blocked, $CASN_i$ would have been able to make an optimal trade-off. Unfortunately, there is no way for $CASN_i$ to have this kind of information.

The terms used in the section are summarized in the following table:

Terms	Meanings
P	the number of processes in the system
N	the number of words needing to be updated atomically
w_i	the number of words currently kept by $CASN_i$
$blocked_i$	the number CASN operations blocked by $CASN_i$
r_i	the average contention on words currently kept by $CASN_i$
m	the lower bound of average contention r_i
M	the upper bound of average contention r_i
c	the competitive ratio

Figure 4: The term definitions

3.1 The First Algorithm

Our first algorithm concentrates on the question of when $CASN_i$ should release all the w_i words that it has acquired. A simple solution is as follows: if the average contention on the w_i words, $r_i = \frac{blocked_i}{w_i}$, is greater than a certain threshold, $CASN_i$ releases all its w_i words to help the *many* other CASNs to go ahead. However, how big should the threshold be in order to optimize the trade-off? In our first reactive CASN algorithm, the threshold is calculated in a similar way to the *reservation price policy* [4]. This policy is an optimal deterministic solution for the online search problem where a player has to decide whether to exchange his dollars to yens at the current exchange rate or to wait for a better one without knowing how the exchange rate will vary.

Let P and N be the number of processes in the system and the number of words needing to be updated atomically, respectively. Because $CASN_i$ only checks the release-condition when: i) it is blocked and ii) it has locked at least a word and blocked at least a CASN, we have that $1 \leq blocked_i \leq (P-2)$ and $1 \leq w_i \leq (N-1)$. Therefore, $m \leq r_i \leq M$ where $m = \frac{1}{N-1}$ and $M = P - 2$. Our *reservation contention policy* is as follows:

Reservation contention policy: $CASN_i$ releases its w_i words when the average contention r_i is greater than or equal to $R^* = \sqrt{Mm}$.

The policy is $\sqrt{\frac{M}{m}}$-competitive. For the proof and more details on the policy, see *reservation price policy* [4].

Beside the advantage of reducing collision, the algorithm also favors CASN operations that are closer to completion, i.e w_i is larger, or that cause a small number of conflicts, i.e. $blocked_i$ is smaller. In both cases, r_i becomes smaller and thus $CASN_i$ is unlikely to release its words.

3.2 The Second Algorithm

Our second algorithm decides not only when to release the $CASN_i$'s words but also how many words need to be released. It is intuitive that the $CASN_i$ does not need to release all its w_i words but releases *just enough* so that most of the CASN operations blocked by $CASN_i$ can go ahead.

The second reactive scheme is influenced by the idea of the *threat-based algorithm* [4]. The algorithm is an optimal solution for the one-way trading problem, where the player has to decide whether to accept the current exchange rate as well as how many of his/her dollars should be exchanged to yens at the current exchange rate.

Definition 3.1. *A* transaction *is the interval from the time a CASN operation is blocked to the time it is unblocked and acquires a new word*

In our second scheme, the following rules must be satisfied in a *transaction.* According to the threat-based algorithm [4], we can obtain an optimal competitive ratio for unknown duration variant $c = \varphi - \frac{\varphi-1}{\varphi^{1/(\varphi-1)}}$ if we know only φ, where $\varphi = \frac{M}{m}$; m and M are the lower bound and upper bound of the average contention on the words acquired by the CASN as mentioned in subsection 3.1, respectively:

1. Release words only when the current contention is greater than $(m * c)$ and is the highest so far.

2. When releasing, release *just enough* words to keep the competitive ratio c.

Similar to the threat-based algorithm [4], the number of words which should be released by a blocked $CASN_i$ at time t is $d_i^t = D_i * \frac{1}{c} * \frac{r_i^t - r_i^{t-1}}{r_i^t - m}$, 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 stands for the average contention on the words kept by a $CASN_i$ and is calculated by the following formula: $r_i = \frac{blocked_i}{w_i}$ as mentioned in Section 3.1. 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 previous 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.

Therefore, beside the advantage of reducing collision, the second algorithm favors to release the words with high contention.

4. Implementations

In this section, we describe our reactive multi-word compare-and-swap implementations.

The synchronization primitives related to our algorithms are *fetch-and-add (FAA)*, *compare-and-swap (CAS)* and *load-linked/validate/store-conditional (LL/VL/SC)*. The definitions of the primitives are described in Figure 5, where x is a variable and v, old, new are values.

```
FAA(x, v)
atomically {
    oldx ← x;
    x ← x + v;
    return(oldx)
    }

CAS(x, old, new)
atomically {
    if(x = old)
      x ← 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 ← v; return(true);
    else return(false);
    }
```

Figure 5: Synchronization primitives

For the systems that support *weak LL/SC* such as the SGI Origin2000 or the systems that support *CAS* such as the SUN multiprocessor machines, we can implement the *LL/VL/SC* instructions algorithmically [13].

4.1 First Reactive Scheme

The part of a CASN operation ($CASN_i$) that is of interest for our study is the part that starts when $CASN_i$ is blocked while trying to acquire a new word after having acquired some words; and ends when it manages to acquire a new word. This word could have been locked by $CASN_i$ before $CASN_i$ was blocked, but 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. The *contention threshold* R^* is computed according to the *reservation contention policy* in Section 3.1. One interesting feature of this reactive helping method is that it favors processes closer to completion as well as processes with a small number of conflicts.

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: i) the identity of helping-process *helping*, ii) the identity of helped-CASN i, iii) the position from which the process will help the CASN lock words *pos*, and iv) the version *ver* of current variable $OP[i]$. In the *Help* procedure, the *helping* process chooses a correct way to help $CASN_i$ according to its state. At the beginning, $CASN_i$'s state is *Lock*. In the *Lock* state, the *helping* process tries to help $CASN_i$ lock all necessary words:

- If the $CASN_i$ 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*).

- If the $CASN_i$, when trying to lock all the N words, 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 $CASN_i$ is blocked by another $CASN_j$, it checks the unlock-condition before helping $CASN_j$ (line 6 in *Locking*). If the unlock-condition is satisfied, $CASN_i$'s state changes into *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*). 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 that other processes whose CASNs are blocked on the words acquired by $CASN_i$

```
type word_type = record value; owner; end; /*normal 32-bit words*/
     state_type = {Lock, Unlock, Succ, Fail, Ends, Endf};
     para_type = record N: integer; addr: array[1..N] of *word_type ; exp, new: array[1..N] of word_type; /*CASN*/
                     state: state_type; blocked : 1..P; end; /*P: #processes*/
     return_type = record kind:{Succ, Fail, Circle}; cId:1..P; end;/*cId: Id of a CASN participating in a circle-help*/
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; /*keeping CASNs currently helped by the process*/
            l: of 1..P; /*the number of CASNs currently helped by the process*/
```

```
/* Version[i] must be increased by one before OP[i] is
used to contain parameters addr[], exp[] and new[] for
Casn(i) */
state_type Casn(i)
begin
1:     OP[i].blocked := 0; OP[i].state := Lock;
       for j := 1 to P do casn_l[j] = 0;
2:     l := 1; casn_l[l] := i;
       /*record CASN_i as a currently helped one*/
3:     Help(i, i, 0, Version[i]);
       return OP[i].state;
end.

return_type Locking(helping, i, pos)
begin
start:
       for j := pos to OP[i].N do
       /*only help from position pos*/
1:         e.addr = OP[i].addr[j]; e.exp = OP[i].exp[j];
       again:
2:         x := LL(e.addr);
3:         if (not VL(&OP[i].state)) then return (nil, nil);
              /*return to read its new state*/
4:         if (x.value ≠ e.exp) then
              return (Fail, nil); /*x was updated*/
           else if (x.owner = nil) then /*x is available*/
5:            if (not SC(e.addr, (e.exp, i)) then goto again;
           else if (x.owner ≠ i) then
           /*x is locked by another CASN*/
              /*check unlock-condition*/
6:            CheckingR(i, OP[i].blocked, j − 1);
7:            if (x.owner in casn_l) then
                 return (Circle, x.owner); /*circle-help*/
8:            Find index k: OP[x.owner].addr[k] = e.addr;
9:            ver = Version[x.owner];
10:           if ( not VL(e.addr)) then goto again;
11:           FAA(&OP[x.owner].blocked, 1);
12:           l := l + 1; casn_l[l] := x.owner;
              /*record x.owner*/
13:           r := Help(helping, x.owner, k, ver);
14:           casn_l[l] := 0; l := l − 1;
              /*omit x.owner's record*/
15:           if ((r.kind = Circle) and (r.cId ≠ i)) then
                 return r; /*CASN_i is not the expected
                                CASN in the circle help*/
              goto start;
       return (Succ, nil);
end.
```

```
return_type Help(helping, i, pos, ver)
begin
start :
1:     state := LL(&OP[i].state);
2:     if (ver ≠ Version[i]) then return (Fail, nil);
       if (state = Unlock) then
       /*CASN_i is in state Unlock*/
3:        Unlocking(i);
          if (helping = i) then
          /*helping is CASN_i's original process*/
4:           SC(&OP[i].state, Lock);
             goto start; /*help CASN_i*/
          else /*otherwise, return to previous CASN*/
5:           FAA(&OP[i].blocked, −1);
5':          return (Succ, nil);
       else if (state = Lock) then
6:        result := Locking(helping, i, pos);
          if (result.kind = Succ) then
          /*Locking N words successfully*/
7:           SC(&OP[i].state, Succ);
             /*change its state to Success*/
          else if (result.kind = Fail) then
          /*Locking unsuccessfully*/
8:           SC(&OP[i].state, Fail);
             /*change its state to Failure*/
          else if (result.kind = Circle) then
          /*the circle help occurs*/
9:           FAA(&OP[i].blocked, −1); return result;
             /*return to the expected CASN*/
          goto start;
       else if (state = Succ) then
10:       Updating(i); SC(&OP[i].state, Ends);
          /*write new values*/
       else if (state = Fail) then
11:       Releasing(i); SC(&OP[i].state, Endf);
          /*release its words*/
       return (Succ, nil);
end.

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

Figure 6: Procedures CASN, Help, Locking and CheckingR in our first reactive multi-word compare-and-swap algorithm

```
UPDATING(i)
begin
     for j := 1 to OP[i].N do
1:       e.addr = OP[i].addr[j]; e.exp = OP[i].exp[j];
2:       e.new = OP[i].new[j];
     again :
3:       x := LL(e.addr);
4:       if (not VL(&OP[i].state)) then return;
           if (x = (e.exp, i)) then
           /*x is expected value & locked by CASN_i */
5:           if ( not SC(e.addr, (e.new, nil)) then
               goto again;
     return;
end.
```

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

Figure 7: Procedures Updating and Unlocking/Releasing in our first reactive multi-word compare-and-swap algorithm

can, on behalf of $CASN_i$, unlock the words and then acquire them while the $CASN_i$'s process helps its blocking CASN operation, $CASN_{x.owner}$ (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 that need to be updated atomically by CASN.

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 CASNs currently blocked by $CASN_i$ and $kept_i$ is the number of words currently locked by $CASN_i$. $CASN_i$ 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*). The unlock-condition is to check whether $\frac{blocked_i}{kept_i} \geq R^*$. Every process blocked by $CASN_i$ on word $OP[i].addr[j]$ increases $OP[i].blocked$ 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 6.

The variable $Version$ is used for memory management 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. 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 8. 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 $Mem[2]$ and then acquire $Mem[5]$. Now, p_2, when helping $CAS3_4$ lock $Mem[5]$, realizes that the word was locked 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.

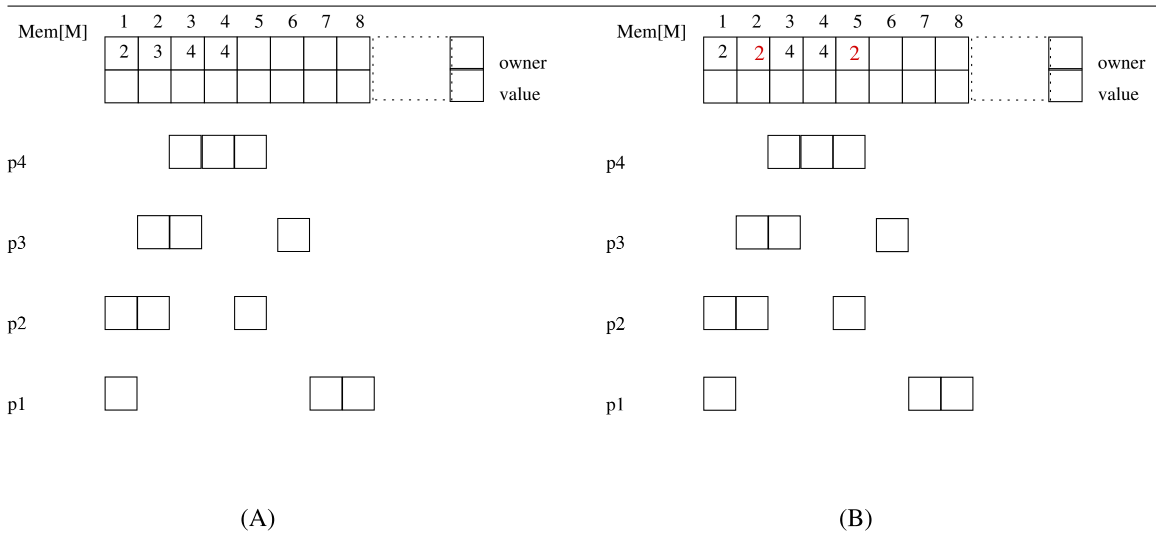

Figure 8: Circle-helping problem: (A) Before helping; (B) After p_1 helps $CAS3_2$ acquire $Mem[2]$ and $Mem[5]$.

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 lth CASN that process p_i is helping at that time. The element is reset to 0 when process p_i completes helping the lth 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*) by setting the returned value $(Circle, x.owner)$ (line 7 in *Locking*). The lth element of the array is set to $x.owner$ before the process helps $CASN_{x.owner}$, its lth 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 that initiated it, p_i, returns and helps it progress (line 4 in $Help$).

4.2 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 CASNs 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 $CASN_i$ is blocked, it only releases *just enough* words to reduce most of CASNs blocked by itself.

According to rule 1 of the second reactive scheme as described in Section 3.2 , 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 9, 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 the second algorithm, the variable $OP[i].blocked$[1] is an array of size N, the number of words need to be updated atomically by CASN. Each element of the array $OP[i].blocked[j]$ is updated in such a way that the number of CASNs blocked on each word is known, and thus a process helping $CASN_i$ can calculate how many words need to be released in order to 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 r_{max} and $init$, respectively. While the $init$ is updated only one time at the beginning of the transaction (line 3 in *CheckingR*), the r_{max} field is updated

1. In our implementation, the array *blocked* is simple a 64-bit word such that it can be read in one atomic step. In general, the whole array can be read atomically by a snapshot operation.

```
type state_type = record init; r_max; ul_pos; state; end;
     para_type = record N: integer; addr: array[1..N] of *word_type ; exp, new: array[1..N] of word_type;
                 state: {Lock, Unlock, Succ, Fail, Ends, Endf}; blocked: array[1..N] of 1..P; end;
                 /* P: #processes; N-word CASN */
```

```
return_type Help(helping, i, pos)
begin
start :
1:   gs := LL(&OP[i].state);
2:   if (ver ≠ Version[i]) then return (Fail, nil);
3:   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.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 = Circle) then
13:        FAA(&OP[i].blocked[pos], −1); return result;
        goto start;
     ...
end.

value_type Read(x)
begin
start :
     y := LL(x);
     while (y.owner ≠ nil) do
          Find index k: OP[y.owner].addr[k] = x;
          ver = Version[y.owner];
          if ( not VL(x)) then goto start;
          Help(self, y.owner, k, ver); y := LL(x);
     return (y.value);
end.
```

```
boolean CheckingR(owner, blocked, kept, gs)
begin
     if (kept = 0) or (blocked = {0..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; /*r is the current contention*/
     if (r < m * C) then return Fail; /* m = 1/(N−1) */
2:   if (kept ≠ gs.ul_pos) then
     /*At the beginning of transaction*/
        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
     /*r is the highest contention so far*/
        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.

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.
```

Figure 9: Procedures Help, Unlocking, CheckingR in our second reactive multi-word compare-and-swap algorithm and the procedure for Read operation

whenever the unlock-condition is satisfied (line 3 and 5 in *CheckingR*). The beginning of a transaction is determined by comparing the number of word currently kept by the CASN, *kept*, and its last unlock-position, *gs.ul_pos* (line 2 in *CheckingR*). The values are different only if the CASN has acquired more words since the last time it was blocked, and thus in this case the CASN is at the beginning of a new transaction according to definition 3.1.

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 it is 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 9.

5. Correctness Proof

In this section, we prove the correctness of our methods. Figure 10 briefly describes the shared variables used by our methods and the 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 10: 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, $CASN_i^j$. The symbol $CASN_i^j$ denotes that this CASN uses the variable $OP[i]$ and that it is the jth time the variable is re-used, i.e. $j = Version[i]$. 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 that p_k read from $OP[i]$, is not correct for p_k.

Lemma 5.1. *Every helping process reads correct values of variable $OP[i]$.*

Proof. In our pseudo-code described in Figure 6, Figure 7 and Figure 9, 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 3 in *Unlocking*, *Releasing*, *Locking* and line 4 in *Updating*). 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 does 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 the 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 that $OP[i].state$ 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 the process that changes the state into $Fail$ reads an unexpected value. The linearization point of *Read* is the point when $y.owner == nil$. It is easy to realize that our specific *Read* operation[2] is linearizable to RCASN. The *Read* operation is similar to those in [8][12].

Now, we prove that the interferences among the procedures do not affect the correctness of our algorithms.

We focus on the changes of $OP[i].state$. We need to examine only four states: $LOCK$, $UNLOCK$, $SUCCESS$ and $FAILURE$. State END is only used to inform whether the CASN has succeeded and it does not play any role in the algorithm. Assume the latest change occurs at time t_0. The processes helping $CASN_i$ are divided into two groups: group two consists of the processes detecting the latest change and the rest are in group one. Because the processes in the first group read a wrong state (it fails to detect the latest change), the states they can read are $LOCK$ or $UNLOCK$, i.e. only the states in phase one.

Lemma 5.2. *The processes in group one have no effect on the results made by the processes in group two.*

Proof. To prove the lemma, we consider all cases where a process in group two can be interfered by processes in group one. Let p_l^j denote process p_l in group j. Because the shared variable $OP[i].block$ is only used to estimate the contention level, it does not affect the correctness of CASN returned results. Therefore, we only look at the two other shared variables Mem and $OP[i].state$.

Case 1.1 : Assume p_k^1 interferes with p_l^2 while p_l^2 is executing procedure $Help$ or $CheckingR$. Because these procedures only use the shared variable $OP[i].state$, in order to interfere with p_l^2 p_k^1 must update this variable, i.e. p_k^1 must also execute one of the procedures $Help$ or $CheckingR$. However, because p_k^1 does not detect the change of $OP[i].state$ (it is a member of the first group), the change must happen after p_k^1 read $OP[i].state$ by LL at line 1 in $Help$, and consequently it will fail to update $OP[i].state$ by SC. Therefore, p_k^1 cannot interfere p_l^2 while p_l^2 is executing procedure $Help$ or $CheckingR$, or in other words, p_l^2 cannot be interfered through the shared variable $OP[i].state$.

2. The *Read* procedure described in figure 9 is used in both reactive CASN algorithms

Case 1.2 : p_l^2 is interfered through a shared variable $Mem[x]$ while executing one of the procedures *Unlocking*, *Releasing*, *Updating* and *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 can only 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 $Mem[x]$. On the other hand, p_k^1 can update $Mem[x]$ only if it is not acquired by another CASN, i.e. $Mem[x] = (OP[i].exp[y], nil)$ or $Mem[x] = (OP[i].exp[y], i)$.

- If p_k^1 wants to update $Mem[x]$ from $(OP[i].exp[y], i)$ to $(OP[i].exp[y], nil)$, the state p_k^1 read is *Unlock*. Because only state *Lock* is the subsequent state from *Unlock*, the correct state p_l^2 read is *Lock*. Because some processes must help the $CASN_i$ successfully release necessary words, which include $Mem[x]$, before the $CASN_i$ could change from *Unlock* to *Lock*, p_k^1 fails to execute $SC(\&Mem[x], (OP[i].exp[y], nil))$ (line 4 in *Unlocking*, Figure 7) and retries by reading $Mem[x]$ again (line 2 in *Unlocking*). In this case, p_k^1 observes that $OP[i].state$ changed and gives up (line 3 in *Unlocking*).
- If p_k^1 wants to update $Mem[x]$ from $(OP[i].exp[y], nil)$ to $(OP[i].exp[y], i)$, the state p_k^1 read is *Lock*. Because the current value of $Mem[x]$ is $(OP[i].exp[y], nil)$, the current state p_l^2 read is *Unlock* or *Failure*.
  - If p_k^1 executes $SC(\&Mem[x], (OP[i].exp[y], i))$ (line 5 in *Locking*, Figure 6) *before* p_l^2 executes $SC(\&Mem[x], (OP[i].exp[y], nil))$ (line 4 in *Unlocking*/*Releasing*, Figure 7), p_l^2 retries by reading $Mem[x]$ again (line 2 in *Unlocking*/*Releasing*) and eventually updates $Mem[x]$ successfully.
  - If p_k^1 executes $SC(\&Mem[x], (OP[i].exp[y], i))$ (line 5 in *Locking*) *after* p_l^2 executes $SC(\&Mem[x], (OP[i].exp[y], nil))$ (line 4 in *Unlocking*/*Releasing*), p_k^1 retries by reading $Mem[x]$ again (line 2 in *Locking*). Then, p_k^1 observes that $OP[i].state$ changed and gives up (line 3 in *Locking*).

Therefore, we can conclude that p_k^1 cannot interfere with p_l^2 through the shared variable $Mem[x]$, which together with case 1.1 results in that the processes in group one cannot interfere with the processes in group two via the shared variables. □

Lemma 5.3. *The interferences between processes in group two do not violate linearizability.*

Proof. 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 result. □

From Lemma 5.2 and Lemma 5.3, we can infer the following corollary:

Corollary 5.1. *The interferences among the procedures do not affect the correctness of our algorithms.*

Lemma 5.4. *A $CASN_i$ blocks another $CASN_j$ at only one position in Mem for all times $CASN_j$ is blocked by $CASN_i$.*

Proof. Assume towards contradiction that $CASN_j$ 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 when $CASN_j$ 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 can only be acquired by one CASN. □

Lemma 5.5. *The algorithms are lock-free.*

Proof. We prove the lemma by contradiction. Assume that no CASN in the system can progress. Because in our algorithms a CASN operation cannot progress only if it is blocked by another CASN on a word, each CASN operation in the systems must be blocked by another CASN on a memory word. Let $CASN_i$ be the CASN that acquired the word w_h with highest address among all the words acquired by all CASNs. Because the N words are acquired in the increasing order of their addresses, $CASN_i$ must be blocked by a $CASN_j$ at a word w_k where $address(w_h) < address(w_k)$. That mean $CASN_j$ acquired a word w_k with the address higher than that of w_h, the word with highest address among all the words acquired by all CASN. This is contradiction. □

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

Lemma 5.6. *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 $CASN_{owner}$ is in a transaction, i.e. being blocked by another CASN, for instance $CASN_j$, the process will certainly call *CheckingR* to check the unlock-condition. Additionally, in our second method the average contention can increase when the CASN releases some of its words and this increase is checked at line 5 in procedure *Help* in figure 9. □

Lemma 5.6 has the important consequence that the process always detects the *average contention on the acquired words* of a CASN whenever it increases, so applying the online-search and one-way trading algorithms with the value the process obtains for the average contention is correct according to the algorithm.

Lemma 5.7. *Procedure CheckingR in the second algorithm computes unlock-point ul_point 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 words can be acquired. □

6. Evaluation

We compared our algorithms to the two best previously known alternatives: i) the lock-free 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. Note that the algorithm in [14] is not intrinsically wait-free because it needs an evaluating function from the user to identify whether the CASN will stop and return when the contention occurs. If we pass the above dummy function to the CASN, the algorithm is completely lock-free.

Regarding the multi-word compare-and-swap algorithm 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 was not available during the time we performed our experiments. However, relying on the experimental data of the paper [8], we can conclude that this algorithm performs approximately as fast as iSTM did in our experiments, in the shared memory size range from 256 to 4096 with sixteen threads.

The system used for our experiments was an ccNUMA SGI Origin2000 with thirty two 250MHz MIPS R10000 CPUs with 4MB L2 cache 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 according to the implementation shown in Figure 5 of [13], where fields *tag* and *val* of *wordtype* were 32 bits each. The experiments were run in 64-bit mode.

The shared memory Mem is divided into N equal parts, and the ith word in N words needing to be updated atomically is chosen randomly in the ith part of the shared memory to ensure that words pointed by $OP[i].addr[1]...OP[i].addr[N]$ are in the increasing order of their indices on Mem. Paddings are inserted between every pair of adjacent words in Mem to put them on separate cache lines. The values that will be written to words of Mem are contained in a two-dimensional 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 the procedure Read, which also uses the 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 thread, the time only used for CASNs. The useful time is calculated by subtracting the overhead time from the total time. The number of successful CASNs is the sum of the numbers of successful CASNs on each thread. Each thread executing the CASN operations precomputes N vectors of random indices corresponding to N words of each CASN prior to the timing test. 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. Figure 11 shows the total number of CASN and the number of *successful* CASN varying with the shared memory size. We think this kind of chart gives the reader a good view on how each algorithm behaves when the contention level varies by comparing the total number of CASN and the number of *successful* CASN.

6.1 Results

The results show that our CASN constructions compared to the previous constructions are significantly faster for almost all cases. The left charts in Figure 11 describes the number of CASN operations performed in one second by the different constructions.

In order to analyze the improvements that are because of the reactive behavior, 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. 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. 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. The left charts in Figure 11 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.

The right charts in Figure 11 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. When the contention is not at its full peak, our reactive schemes catch up fast and help the CASN operations 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 the left charts of Figure 11:

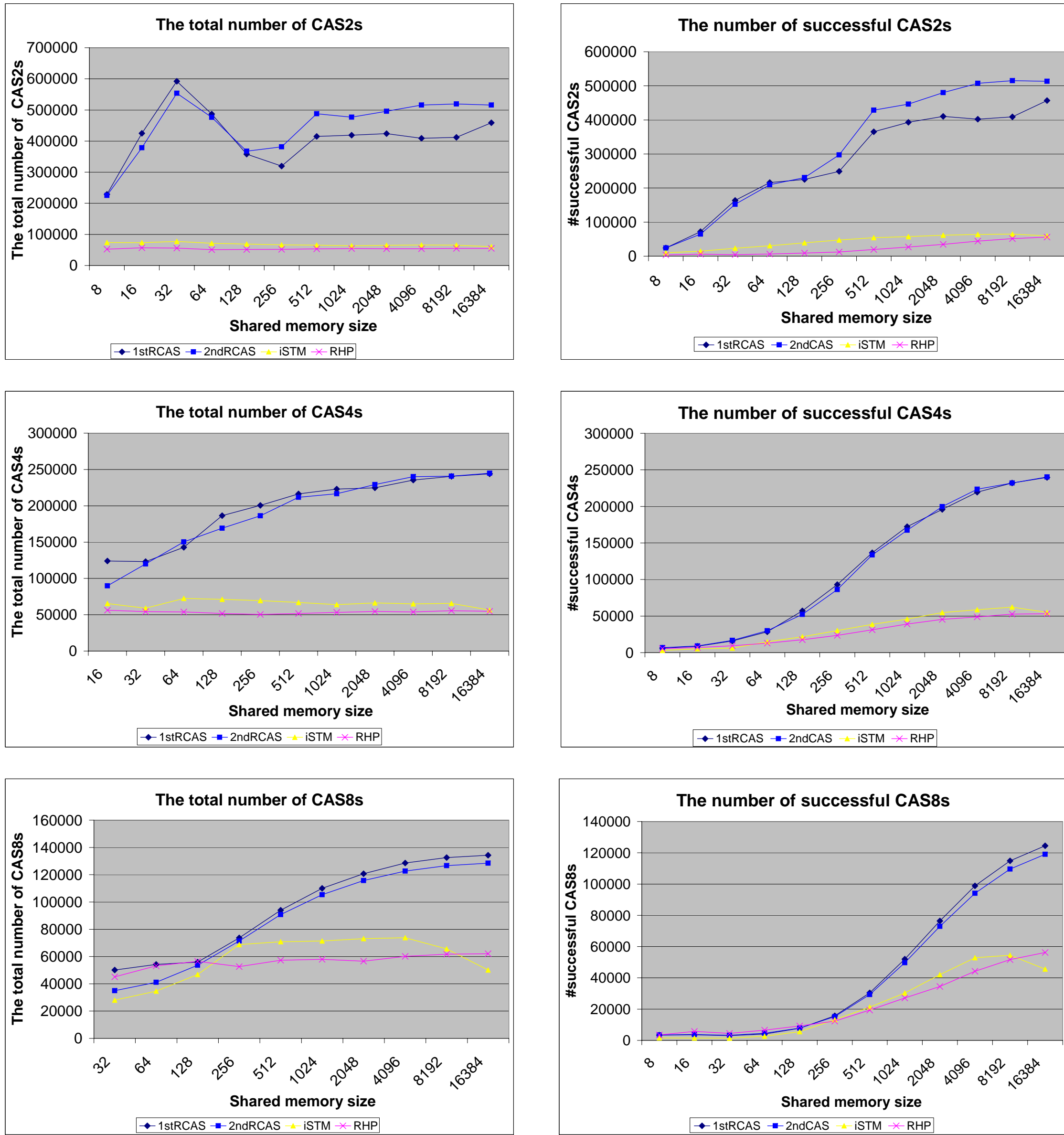

Figure 11: The numbers of CAS2s, CAS4s and CAS8s and the number of *successful* CAS2s, CAS4s and CAS8s in one second

CAS2 : the first reactive compare-and-swap (1stRCAS) and the second one (2ndRCAS) are about seven times and nine times faster than both RHP and iSTM, respectively.

CAS4 : both RCAS are four times faster than both RHP and iSTM.

CAS8 : both RCAS are two times faster than both RHP and iSTM.

Regarding the number of successful CASN operations, our RCAS algorithms still outperform RHP and iSTM in almost all cases. Similar to the above results, at the memory size of 16384 in the right charts of Figure 11, both reactive compare-and-swap operations perform faster than RHP and iSTM from two to nine times.

7. Conclusions

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 contention level 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. Both our algorithms are linearizable. Experiments on thirty processors of an SGI Origin2000 multiprocessor show that both our algorithms react quickly according to the contention conditions and significantly outperform the best-known alternatives in all contention conditions.

In the near future, we plan to look into new reactive schemes that may further improve the performance of reactive multi-word compare-and-swap implementations. The reactive schemes used in this paper are based on competitive online techniques that provide good behavior against a malicious adversary. In the high performance setting, a weaker adversary model might be more appropriate. Such a model may allow the designs of schemes to exhibit *more active* behavior, which allows faster reaction and better execution time.

Acknowledgements

The authors wish to thank Marina Papatriantafilou and Håkan Sundell for their assistance during the progress of this work. We also thank our anonymous reviewers for their helpful comments on the presentation of this paper.

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, June 1993.

[4] R. El-Yaniv, A. Fiat, R. M. Karp, and G. Turpin, "Optimal Search and One-Way Trading Online Algorithms", *Algorithmica 30*, Springer-Verlag, pp. 101–139, 2001.

[5] M. Greenwald and D.R. Cheriton, "The Synergy Between Non-blocking Synchronization and Operating System Structure", *Proceedings of the Second Symposium on Operating System Design and Implementation*, USENIX, Seattle, pp 123-136, October 1996.

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

[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, K. Fraser and I. A Pratt, "A practical multi-word compare-and-swap operation", *Proceedings of the 16th International Symposium on Distributed Computing*, Springer-Verlag, pp. 265–279, October 2002.

[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)*, pp. 745–770, November 1993.

[11] M. Herlihy, "Wait-Free Synchronization", *ACM Transactions on Programming Languages and Systems 11(1)*, pp. 124–149, January 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", *Proceedings of 11th International Workshop on Distributed Algorithms, LNCS1320*, Springer-Verlag, pp. 305–319, 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 29(3)*, Kluwer Academic, pp. 249–282, 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)*, Springer-Verlag, March 2002.

[18] P. Tsigas, Y. Zhang, “Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors”, *Proceedings of the ACM SIGMETRICS 2001/Performance 2001*, pp. 320–321, June 2001.

[19] P. Tsigas, Y. Zhang, “Integrating Non-blocking Synchronization in Parallel Applications: Performance Advantages and Methodologies” *Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP’02)*, pp. 55–67, July 2002.

[20] The manpages of SGI Origin 2000.