A Practical Multi-Word Compare-and-Swap Operation

Timothy L. Harris, Keir Fraser and Ian A. Pratt

University of Cambridge Computer Laboratory, Cambridge, UK
{tim.harris,keir.fraser,ian.pratt}@cl.cam.ac.uk

Abstract. Work on non-blocking data structures has proposed extending processor designs with a compare-and-swap primitive, CAS2, which acts on two arbitrary memory locations. Experience suggested that current operations, typically single-word compare-and-swap (CAS1), are not expressive enough to be used alone in an efficient manner. In this paper we build CAS2 from CAS1 and, in fact, build an arbitrary multi-word compare-and-swap (CASN). Our design requires only the primitives available on contemporary systems, reserves a small and constant amount of space in each word updated (either 0 or 2 bits) and permits non-overlapping updates to occur concurrently. This provides compelling evidence that current primitives are not only universal in the theoretical sense introduced by Herlihy, but are also universal in their use as foundations for practical algorithms. This provides a straightforward mechanism for deploying many of the interesting non-blocking data structures presented in the literature that have previously required CAS2.

1 Introduction

CASN is an operation for shared-memory systems that reads the contents of a series of locations, compares these against specified values and, if they all match, updates the locations with a further set of values. All this is performed atomically with respect to other CASN operations and specialized reads. The implementation of a non-blocking multi-word compare-and-swap operation has been the focus of many research papers [7, 10, 2, 3, 15, 5]. As we will show none of these provides a solution that is practicable in terms of the operations it requires from the processor, its storage costs and the features it supplies.

This paper presents a new design that solves these problems. The solution is valuable because it finally allows many algorithms requiring CASN to be used in earnest (for example those from [11, 5, 4]). CASN is useful as a foundation for building concurrent data structures because it can update a set of locations between consistent states. Aside from its applicability, our solution is notable in that it considers the full implementation path of the algorithm. Previous work has often needed a series of abstractions to build strong primitives from those actually available – each layer adds costs, the sum of which places the algorithm beyond reasonable use.

We present our new design through a two-stage process: we develop a restricted form of CAS2 directly from CAS1 (Sect. 4) and we then show how to use that to implement CASN (Sect. 5). In Sect. 6 we discuss implementation problems such as memory management and the need for memory barrier operations. We evaluate the algorithm through experimental results on six processor families.

2 Background

Throughout this paper we assume a shared-memory model. We assume that an operation new allocates a fresh area of memory sufficient for a specified number of words. We take CAS1 as a primitive and assume initially that it – along with ordinary read and write operations – is implemented in a *linearizable* manner by the system (meaning that it appears to occur atomically at some point between its invocation and return). We assume that all memory accesses are of word-size and are to word-aligned addresses. As usual we define CAS1 as:

```
word_t CAS1(word_t *a, word_t o, word_t n)
{ old = *a;
  if (old == o) *a = n;
  return old;
}
```

We wish CASN to be *linearizable* so that it is easy to reason about its use. It should be *non-blocking*, meaning that some operation will complete if the system takes a large enough finite number of steps. This gives resilience against poor scheduler interactions (e.g. priority inversion). For scalability it is crucial that it is *disjoint-access-parallel*: operations on disjoint sets of locations should proceed in parallel. Finally, it should act on data structures with a natural and efficient representation. This means that reserving more than a few bits in each location is unreasonable. We would like to be able to use a built-in CAS1 operation for the case of a single-word update. It is usually necessary to use separate read and write operations on locations subject to update by CASN since disjoint-access-parallel designs place intermediate values in locations during their update.

2.1 Related Work

Herlihy's universal construction may form the basis of a CASN design, but it is not disjoint-access-parallel [7]. Neither is Greenwald's basic implementation using CAS2 [5], although he also shows how a further control word per word allows parallel updates. Israeli and Rappaport's design is disjoint-access-parallel [10], but each word must hold a processor 'ownership' field and the algorithm requires strong LL/SC operations. Those operations can be implemented over basic LL/SC or CAS1 by reserving further per-processor 'valid' bits in each word.

Anderson and Moir's wait-free CASN uses strong LL/SC single-word primitives [2]. It requires extensive auxiliary per-word structures. Moir subsequently developed a simpler *conditionally wait-free* design for CASN [15], meaning one

Table 1. CASN algorithms for a system with p processes, a machine word size of w bits, a maximum CASN width of n locations from a addresses, showing which algorithms are disjoint-access-parallel (*D-A parallel*) and which require support from the operating system kernel (*OS*)

	D-A parallel	Requires	Bits-per-word
[7]	No	`CAS1`	0
[5]	No	`CAS2`	0
[3]	No	`CAS1` + OS	0
[2]	Yes	Strong `LL/SC`	$p(w+l)+l$ where $l = \lg_2 p + \lg_2 a$
[15]	Yes	Strong `LL/SC`	$\lg_2 p + \lg_2 n$
[10]	Yes	Strong `LL/SC`	$\lg_2 p$
[3]	Yes	`CAS1` + OS	$1 + \lg_2 n + \lg_2 p$
[10]	Yes	`CAS1`	$p + \lg_2 p$
New	Yes	`CAS1`	0 or 2

that is not intrinsically wait-free but which, at key points after contention is detected, evaluates a user-supplied function to determine whether to retry.

Anderson *et al.* provide two further algorithms for priority-based schedulers [3]. One is only suitable for uniprocessors. The other is not disjoint-access-parallel. In their designs Anderson *et al.* use a restricted form of CAS2 which is much the same as the RDCSS operation we define in Sect. 4. However, it requires priority-based scheduling and non-preemption guarantees for some code sequences.

Moir shows several ways to build strong LL/SC from CAS1 or realistic LL/SC [14]. However, there are problems with each construction. The correctness of the first relies on sufficiently large counters reserved in each value not overflowing at certain points. The second design allows ponter-sized values to be stored by fragmenting them across words along with a header. This (at least) doubles the storage required. The third design provides single-word LL/SC operations without needing to avoid overflow based on an elaborate mechanism to control tag re-use – for example a single SC requires four operations on a tag-management queue. This design also requires a processor ID field to be reserved in every word, along with space for these bounded tags and a count field. None of these algorithms fits with our desire for a natural and efficient representation.

Table 1 summarizes the various existing CASN designs and contrasts them with our algorithm. For all except [2] and [15] the per-word overhead is reserved in each data location; in [2] and [15] it is separate.

3 Algorithmic Overview

As with most concurrent algorithms, the design of ours is rather intricate. We hope that a brief overview of the the algorithm's operation will aid readability. Central to it is the use of *descriptors*. These are data structures in which threads initiating some operation make available all of the information that others need to complete it – e.g. a `CASN` descriptor holds the addresses to be updated, the values expected to be found there, the new values to store and a status field indicating whether the `CASN` is still in progress.

A thread makes a descriptor active by placing a pointer to it into a location in shared memory. This is our non-blocking alternative to locking that location. Other threads seeing the descriptor pointer use the information in it to help the owning thread complete its operation and release the location. A `CASN` proceeds by placing pointers to its descriptor in each location being updated, checking that they hold the expected old values. If this succeeds for all the locations then each location is released, replacing the descriptor-pointers with the new values. If any location does not hold the requisite old value then the `CASN` is said to have failed and each location is restored to its old value. `CASN` therefore resembles an update made using two-phase locking, but employing descriptor pointers so that other threads accessing the locations do not block.

We decompose `CASN` into two layers. We first build a limited form of `CAS2` (Sect. 4) that atomically introduces or removes descriptor-pointers conditional on a status field. From this we construct `CASN` (Sect. 5). Sect. 6 considers implementation issues and the management of the memory holding descriptors.

4 Double-Compare Single-Swap

We define `RDCSS` as a restricted form of `CAS2` operating atomically as:

```
word_t RDCSS(word_t *a1, word_t o1, word_t *a2, word_t o2, word_t n2)
{ r = *a2;
  if ((r == o2) && (*a1 == o1)) *a2 = n2;
  return r;
}
```

This is *restricted* in that (*i*) only the location `a2` can be subject to an update, (*ii*) the memory it acts on must be partitioned into a *control section* (within which `a1` lies) and a *data section* (within which `a2` lies), and (*iii*) the function returns the value from from `a2` rather than an indication of success or failure. `RDCSS` may operate concurrently with (*i*) any access to the control section, (*ii*) reads from the data section using `RDCSSRead`, (*iii*) other invocations of `RDCSS` and (*iv*) updates to the data section using `CAS1`, subject to the constraint that such `CAS1` may fail if an `RDCSS` operation is in progress on that location.

4.1 Design

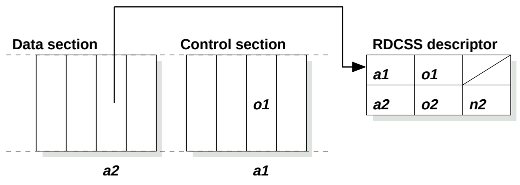
Figure 1 shows pseudo-code to implement `RDCSS` from `CAS1`. The *descriptor* passed to `RDCSS` contains five fields defining the proposed operation: the *control address* (`a1`), *expected value* (`o1`), *data address* (`a2`), *old value* (`o2`) and *new value* (`n2`). Descriptors are held outside the control and data sections and (aside from those introduced by `RDCSS`) values in the data section are distinct from pointers to descriptors. Each invocation uses a fresh descriptor, meaning one whose

```
word_t RDCSS (RDCSSDescriptor_t *d) {
  do {
    r = CAS1(d->a2, d->o2, d); /* C1 */
    if (IsDescriptor (r)) Complete(r); /* H1 */
  } while (IsDescriptor (r)); /* B1 */
  if (r == d->o2) Complete(d);
  return r;
}

word_t RDCSSRead (addr_t *addr) {
  do {
    r = *addr; /* R1 */
    if (IsDescriptor(r)) Complete(r); /* H2 */
  } while (IsDescriptor (r)); /* B2 */
 return r;
}

void Complete (RDCSSDescriptor_t *d) {
  v = *(d->a1); /* R2 */
  if (v==d->o1) CAS1(d->a2, d, d->n2); /* C2 */
    else CAS1(d->a2, d, d->o2); /* C3 */
}
```

Fig. 1. RDCSS pseudo-code implementation

address is (or acts as if it is) held only by the caller. A predicate `IsDescriptor` tests whether its parameter points to a descriptor – we discuss it in Sect. 6.

In outline `RDCSS` attempts a `CAS1` on the data address to change the old value into a pointer to the descriptor (`C1`). If successful, `Complete` finishes the operation: if the control address holds the expected value then the pointer is changed to the new value (`C2`), otherwise the old value is re-instated (`C3`). If a descriptor is found (`H1`, `H2`) then that `RDCSS` invocation is completed. A descriptor is 'active' when referenced from the data section, for example:



4.2 Correctness

We wish to establish that `RDCSS` and `RDCSSRead` provide linearizable non-blocking implementations. We proceeded by developing a model from the pseudo-code definitions and subjected this to exhaustive tests using the Spin model checker [9]. Direct model checking is impracticable: the size of the shared memory, the number of active threads and the number of concurrent `RDCSS` invocations are unbounded. However, inspection of the algorithm lets us reduce the size of the state space to one which can be explored successfully:

- Each invocation of `RDCSS` can be considered separately. This surprising observation follows by examining the memory accesses. `C1` is the only one to make a descriptor active and it succeeds at most once per descriptor (its return value causes loop `B1` to terminate). Updates, `C2` and `C3` make descriptors inactive. Therefore each descriptor has at most one interval of time over which it is active, causing at most two updates – one to active it and one to deactivate it. Both updates are to the data address specified in the descriptor. Different `RDCSS` operations acting on the same location are thereby serialized by the order of their active periods, so we can consider them individually.
- We divide values in memory into equivalence classes. We classify the contents of the control address as either equal or not equal to the expected value. We need four classes for the data address: the old value, the new value, a pointer to the descriptor active on it and finally all other values.
- Although there may be an unbounded number of threads in the system, each is in one of a limited number of states defined by a point in the code of Fig. 1 and the values of local variables. We model the threads collectively as a set of pairs (p, m) where p represents a possible thread state and m is a boolean indicating whether at most one, or possibly more than one, thread is in that state. For example, this set initially contains one pair representing a single thread invoking `RDCSS` and multiple potential invocations of `RDCSSRead`.

We hypothesized that `RDCSS` can be linearized at the last execution of `R2` for a descriptor that becomes active and otherwise at the last execution of `C1`. `RDCSSRead` would be linearized at its last execution of `R1`. From this abstraction we developed a Spin model in which global variables represent (*i*) the set of possible thread states (*ii*) the contents of the control and data addresses and (*iii*) the 'logical' contents of the data address, updated at the proposed linearization point of `RDCSS` and read at the proposed linearization point of `RDCSSRead`.

We model execution by defining a guarded statement for each thread state, enabled when that state is possible. Additional statements, always enabled, model operations that can operate concurrently with `RDCSS`– for example external updates to the value held at the control address. We used assertion statements to compare the logical and actual memory contents at the proposed linearization points. Spin accepts the resulting model without any assertion failures.

Showing non-blocking behaviour proceeds more directly: observe that each backward branch (`B1`, `B2`) is taken only if a descriptor-pointer was read from the data section and `Complete` invoked on that descriptor. Each descriptor-pointer is stored in the data section at most once (as above, at `C1`) and each invocation of `Complete` either removes the descriptor-pointer (if `C2` or `C3` succeeds) or observes it to have already been removed (if the attempted `CAS1` fails. Therefore backward branches can only occur if system-wide progress has been made.

```
bool CASN (CASNDescriptor_t *cd) {
  if (cd->status == UNDECIDED) { /* R4 */
phase_1: status = SUCCEEDED;
    for (i = 0; (i < cd->n) && (status == SUCCEEDED) ; i++) { /* L1 */
retry_entry: entry = cd->entry[i];
          val = RDCSS (new RDCSSDescriptor_t (&(cd->status), UNDECIDED,
                                        entry->addr, entry->old, cd)); /* X1 */
          if (IsCASNDescriptor_t (val)) {
            if (val != cd) {
              CASN (val); /* H3 */
              goto retry_entry;
            }
          } else if (val != entry->old) status = FAILED;
        }
    CAS1 (&(cd->status), UNDECIDED, status); /* C4 */
  }
phase_2: succeeded = (cd->status == SUCCEEDED);
  for (i = 0; i < cd->n; i ++)
    CAS1 (cd->entry[i].addr, cd,
        succeeded ? (cd->entry[i].new) : (cd->entry[i].old)); /* C5 */
  return succeeded;
}

word_t CASNRead (addr_t *addr) {
  do {
    r = RDCSSRead(addr); /* R5 */
    if (IsCASNDescriptor (r)) CASN (r); /* H4 */
  } while (IsCASNDescriptor (r)); /* B3 */
  return r;
}
```

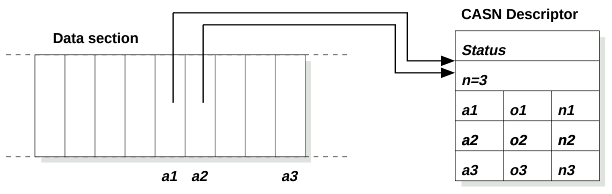
Fig. 2. Two-phase CASN pseudo-code implementation using RDCSS at `X1`

5 CASN Using RDCSS

We will now show how `CASN` can be implemented using `RDCSS`. As before a descriptor held in shared memory describes the operation. A *CASN-descriptor* contains a *status* field (holding `UNDECIDED`, `FAILED` or `SUCCEEDED`), a *count* (`n`) and then a series of `n` *update entries* each having a distinct *update address* (`a1`, . . .), an *old value* (`o1`, . . .) and a *new value* (`n1`, . . .).

The update addresses lie in the data section of memory and are held according to some total order agreed by all threads to guarantee non-blocking behaviour, e.g. sorted. The `CASN` descriptors themselves are held in the control section: the *status* field will be subject to comparison using `RDCSS`. As before, each invocation is made with a fresh descriptor. `CASN` may operate concurrently with (*i*) other invocations of `CASN` and (*ii*) reads from the data section using `CASNRead`.

Fig. 2 shows the two-phased pseudo-code for our CASN algorithm and for an associated CASNRead operation. The first phase attempts to introduce pointers from each update address to the descriptor. For example, after installing two such pointers the memory may be depicted:



If phase 1 encounters a pointer to another descriptor then it helps that operation before re-trying. At the end of phase 1, C4 tries to set the status field to SUCCEEDED (if pointers were installed at each address) or FAILED (if some address did not contain the value expected). The second phase iterates over the update entries removing the pointers. A descriptor is *undecided* whenever its status field holds that value; otherwise it is *decided* and either *failed* or *succeeded.* The *logical contents* of a data location are (*i*) the value it holds if that is not a descriptor pointer, (*ii*) the old value for that location in an undecided or failed descriptor it points to, or (*iii*) the new value for that location in a succeeded descriptor.

CASNRead is structured in the same way as RDCSSRead: it retries the read operation until it does not encounter a descriptor pointer. Although we do not show them here, other kinds of read operation are also possible. One alternative is for CASNRead not to help other CASN invocations and instead to derive the logical contents of the location using the descriptor that it encounters. As Moir observed, the ability to read without helping can aid performance [15].

5.1 Correctness

We initially developed CASN in concert with a Spin model of its behaviour parameterized on the number of concurrent operations, the number of storage locations and the range of values that those locations could hold. The model maintained the actual contents of those locations (updated using our algorithm with CAS1 as a primitive) and the logical contents (updated by an atomic step at the proposed linearization point). The largest configuration that could be checked exhaustively comprised 3 concurrent CAS2 operations on up to 4 binary locations.

While invaluable in identifying problems with early designs, this approach also helped us develop our ideas of why the algorithm works in a general setting. In this section we show that CASN is linearizable, performing an atomic update to the logical contents of memory at the point the descriptor becomes decided.

Conceptually, the argument is simplified if you consider the memory locations referred to by a particular CASN descriptor and the updates that various threads make on those locations within the implementation of the CASN function. A descriptor's lifecycle can be split into a first *undecided* stage and a second *decided* stage, joined by C4 which updates the descriptor's status field. We show that, aside from `C4`, all of the updates preserve the logical contents of memory and we then show that, when `C4` is executed, it atomically updates the logical contents of the locations on which the `CASN` is acting.

Firstly, we consider the descriptor lifecycle and the updates that can be made by threads operating on undecided descriptors:

Lemma 1. *Descriptor lifecycle is undecided $\rightarrow$ decided.* The only update to the status field is `C4` which specifies `UNDECIDED` as the expected old value and either `FAILED` or `SUCCEEDED` as the new value. The old and new values differ and so `C4` can succeed only once for each descriptor.

Lemma 2. *Threads operating on a descriptor in the undecided stage are in phase 1 of the algorithm.* To reach phase 2 a thread must either complete `C4` (the first to do so would make the descriptor *decided*) or observe the descriptor to be decided at `R4`. In each case a contradiction.

Lemma 3. *Threads operating on undecided descriptors preserve the logical contents of memory.* The only update made in phase 1 of the algorithm is `X1` which replaces the expected old value for a location with a pointer to a `CASN` descriptor specifying that same old value for that location. Hence the logical contents are preserved.

Secondly, we consider the linearization of `CASN` operations:

Lemma 4. *Linearization of failed* `CASN`*.* If `C4` sets the status `FAILED` then `X1` returned a non-descriptor value, not matching the expected old value. The time the unexpected value was read can be taken as the linearization point.

Lemma 5. *Linearization of successful* `CASN`*.* If `C4` sets the status `SUCCEEDED` then the `L1` completed and so (*i*) for each update entry, either `X1` installed a descriptor-pointer or it found a pointer already in place, (*ii*) those values remain in place since no thread is yet in phase 2 (Lemma 2) and so (*iii*) `C4` changes the logical contents of all update addresses, forming the linearization point.

Thirdly, we consider the *decided* stage of a descriptor's lifecycle:

Lemma 6. *Threads with visible effects operating on decided descriptors are in phase 2 of the algorithm.* The only updates to shared storage outside phase 2 are `X1` and `C4`. Each tests the descriptor status for `UNDECIDED`.

Lemma 7. *Threads with visible effects operating on decided descriptors preserve the logical contents of memory.* The only update is `C5` and, if it succeeds, the computed value matches the logical contents of the location.

Lemma 8. *All descriptor-pointers are removed after one thread exits phase 2 of the algorithm.* During phase 2 a thread attempts `C5` for each update entry. Only `C5` changes a descriptor-pointer into a non-descriptor pointer: it will fail if (*i*) a descriptor-pointer was not installed at that location or (*ii*) another thread has already removed the descriptor-pointer by its own execution of `C5`.

Finally, a technical requirement of the restrictions `RDCSS` imposes (Sect. 4):

Lemma 9. `CAS1` *operations do not fail because of concurrent* `RDCSS`*.* `C4` acts on the control section so cannot encounter an `RDCSS` descriptor pointer. The only other consideration is between `C5` and `X1`. `C5` has a `CASN` descriptor pointer as its old value and `X1` has a non-descriptor value as its old value, so if `C5` fails because it encounters a pointer to an `RDCSS` descriptor then it would have failed anyway.

6 Implementation

The pseudocode makes a number of assumptions about the underlying platform, and ignores four important problems: allocating and de-allocating descriptors, implementing the `IsDescriptor` predicates, the availability of `CAS1` as an atomic primitive and the non-linearizability of the processor-supplied read/write and `CAS1` primitives. We address those concerns in Sects 6.1–6.4.

6.1 Storage Management

For both `RDCSS` and `CASN` we assumed that fresh descriptors would be used on each invocation. We developed two techniques to reduce allocations. Firstly, we embed a group of `RDCSS` descriptors into each `CASN` descriptor to form a *combined descriptor*. Rather than holding the five `RDCSS` fields directly, these embedded descriptors contain a single constant reference to the enclosing `CASN` descriptor from which the `RDCSS` descriptor values are derived. Secondly, in each combined descriptor, we provide only one embedded `RDCSS` descriptor per thread. This still acts 'as if' fresh addresses are used because (*i*) the addresses operated on by a particular `CASN` are distinct from one another, and (*ii*) the `RDCSS X1` will install a thread's embedded descriptor at most once at each address (if the `RDCSS` does install the pointer then either the loop advances to the next iteration or it terminates because the status field is no longer `UNDECIDED`).

We evaluated two ways of managing these combined descriptors. In the first we assume garbage collection is already provided. In the second we introduce reference counting following Valois' `CAS1`-based design [17]. We use per-thread lists of free descriptors so that, without contention, a descriptor retains affinity for a particular thread. Manipulating reference counts and free lists forms around 10% of the execution time of an un-contended `CASN`. Although this scheme does not allow the storage that holds descriptors to be re-used for other non-reference-counted purposes, it is easy to imagine hybrids in which long-term shrinking uses garbage collection but short-term re-use employs counts. We are currently evaluating such combinations as well as Michael's SMR algorithm [13] and Herlihy *et al.*'s solution to the *Repeat Offender Problem* [8].

6.2 Descriptor Identification

The `IsDescriptor` and `IsCASNDescriptor` predicates must identify pointers to the descriptors used by `RDCSS` and `CASN`. If run-time type information is available then this could be exploited without further storage cost. Otherwise, the pointers themselves can be made distinct by non-zero low-order bits (as we did in previous work to indicate deleted items [6]). We need two bits to distinguish ordinary pointers, references to `RDCSS` descriptors and references to `CASN` descriptors.

We favour this second scheme because it is widely applicable and it avoids an additional memory access to obtain type information. An attractive hybrid scheme, which we have not yet evaluated, is to reserve a single bit to identify descriptor-pointers in general and then to use type information, or prescribed header values, to distinguish between the two kinds.

Table 2. CPU microseconds per successful `CASN` operation on a range of popular four-processor systems and `CASN` widths of 2, 4, 16 and 64 words

	IA-32				**IA-64**				**Alpha**				**SPARC**			
Type	2	4	16	64	2	4	16	64	2	4	16	64	2	4	16	64
HF	2.4	4.2	21	280	1.6	2.8	17	280	2.0	3.9	24	200	3.1	5.5	30	430
HF-RC	2.1	3.6	19	270	1.5	2.6	16	270	2.2	3.7	23	200	3.2	5.5	28	400
IR	4.0	6.3	26	340	3.4	4.4	19	300	4.5	6.4	31	490	5.4	8.7	44	570
MCS	4.8	7.2	22	84	5.6	8.2	24	92	7.1	7.4	17	63	10	16	61	250
MCS-FG	2.1	4.2	17	130	1.4	2.8	14	130	2.6	5.3	26	210	3.5	6.9	43	290

6.3 Atomic Hardware Primitives

Rather than implementing `CAS1` directly, some processors provide the more expressive `LL`/`SC` (load-linked, store-conditional) operations. Unlike the *strong* `LL`/`SC` operations sometimes used in algorithms, these must form non-nesting pairs and `SC` can fail *spuriously* [7].

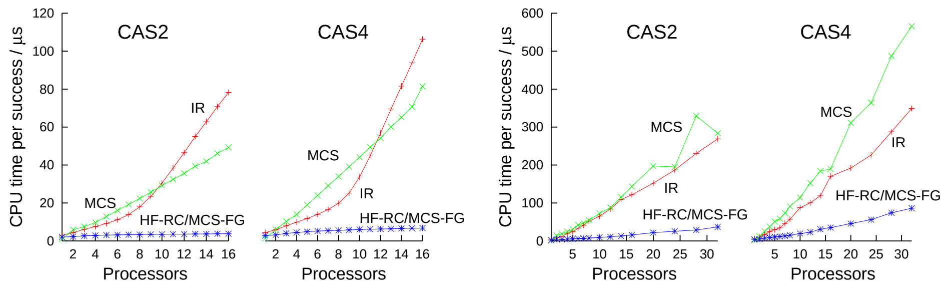
Where necessary we used a software version of `CAS1`, based on `LL`/`SC`, to generate the `CASN` results in this paper. Methods for building stronger primitives from `LL`/`SC` are well known: for example, the Alpha processor handbook shows how to use them to construct atomic single-word read-modify-write sequences such as `CAS1`. Such constructions, based on a simple loop that retries a `LL`/`SC` pair, are non-blocking under a guarantee that there are not infinitely many spurious failures during a single `CAS1` operation.

6.4 Weak Memory Architectures

Finally, our pseudo-code assumes the sub-operations it uses are themselves linearizable. This is not true of modern systems – including all those used in Sect. 7. In general these systems provide cache-coherence, serializability of accesses to single words and total ordering between accesses from the same processor to the same location. Stronger ordering must be established using *barrier* instructions: operations before the barrier must commit before any later operation may be executed. Adve and Gharachorloo provide a tutorial on the subject [1].

7 Evaluation

Our benchmark is much the same as the *resource allocation* one used by Shavit and Touitou's which they argue is representative of highly concurrent queue and counter implementations [16]. A shared vector is initialized with distinct pointers. Each processor loops selecting a set of locations, reading their current values using `CASNRead` and attempting a `CASN` operation to permute the values between the locations. For a test using `CASN` operations of width n we divide the vector into n equal sized buckets and select one entry from each bucket. This benchmark enables a range of contention levels to be investigated by varying the



(*a*) 1–16 Power3-II processors (*b*) 1–32 MIPS processors

Fig. 3. CPU time per successful `CASN` operation for two large systems

concurrency, vector size, the width of `CASN` performed and whether padding is inserted between elements to place them on separate cache lines.

We implemented three non-blocking algorithms: ours assuming a garbage collector (HF) or using reference counting (HF-RC) and Israeli and Rappoport's `CAS1`-based design as the only practical alternative from Fig. 1 (IR). We also implemented two lock-based schemes using the queued spin-lock design of Mellor-Crummey and Scott [12], either with one lock to protect the entire vector (MCS) or with fine grain (i.e. per-entry) locks (MCS-FG).

Our measurements exclude initialisation. We start timing after all threads signal that they are executing and then run for two seconds. Results showing time per successful operation are calculated by dividing the total CPU time used (excluding initialisation) by the number of successful `CASN` operations. The CPU time and successful operations are summed across all threads. All results are presented to 2 significant figures. Although we do not analyse the costs of `CASNRead` operations in isolation, it is worth noting that a well-engineered implementation for any of the non-blocking algorithms adds only two operations to each read from a location that may be subject to `CASN` updates.

7.1 Small Systems

We ran our benchmark application using four threads on four-processor IA-32 Pentium-III, IA-64 Itanium, Alpha 21264 and SPARC Ultra-4 workgroup servers. We used a vector of 1024 elements without padding – in doing so we aim to produce a worst-case layout. The CPU requirements in μs per successful `CASN` are shown in Table 2 over a range of `CASN` widths on each system.

Our `CASN` algorithm performs universally better than the IR scheme. This is the case for every test we ran and follows intuition: the algorithms use the same helping strategy and, for an uncontended n way operation, HF performs $3n + 1$ word-size `CAS1` steps whereas IR performs $4n + 4$ double-width steps. There is little difference in performance between HF and HF-RC: in low contention the

Table 3. Power3-II system (top) and MIPS R12000 system (bottom) : CPU microseconds per successful `CASN` operation with 16 threads *vs.* vector size and `CASN` width

	CAS2			CAS4			CAS8		
Type	256	1024	4096	256	1024	4096	256	1024	4096
HF-RC	4.7	3.8	3.6	11	6.8	5.9	39	17	12
IR	89	79	76	140	110	94	270	160	120
MCS	50	51	49	77	77	78	130	130	130
MCS-FG	4.2	3.6	3.6	11	7.6	6.9	35	18	14
DUMMY	2.8	2.7	2.7	4.8	4.4	4.3	9.4	8.0	7.7

	CAS2			CAS4			CAS8		
Type	256	1024	4096	256	1024	4096	256	1024	4096
HF-RC	21	16	15	55	33	28	180	99	58
IR	130	120	120	190	150	140	470	220	180
MCS	130	120	130	220	190	200	380	380	430
MCS-FG	18	14	12	38	29	24	115	69	54
DUMMY	14	11	12	26	23	21	62	44	40

cost of reference counting is balanced by the locality gained by re-use and as contention rises the main loop of the `CASN` dominates execution.

Only the non-blocking algorithms experience `CASN` failures because the lock-based designs prevent updates between the old values being read and the `CASN` being attempted. On the 2-processor system the non-blocking algorithms exhibit indistinguishable success rates: 97-99% for widths of up to 8, 90% for 16, 70% for 32 and 50% for 64. On the 4-processor machines success rates of 90% and above are achieved for `CASN` widths of 2, 4 or 8.

7.2 Large Systems

We now examine larger systems: an IBM SP node of 16 Power3-II processors with uniform memory access and a ccNUMA Origin 2000 system with 64 MIPS R12000 processors. For these systems we inserted padding between vector elements to place each on its own cache line and eliminate *false sharing*. This improved the performance of all algorithms, particularly where contention was high. Furthermore, we maximized the performance of the MCS-FG algorithm by locating each vector element and its associated lock in the same cache line.

Fig. 3 shows how the CPU time per successful `CASN` varies with the number of processors used. In each case we examined `CAS2` and `CAS4` operating on a vector of 1024 pointers, corresponding to a minimum success rate of 92% on the Power3-II machine and 90% on the MIPS machine. We did not run experiments with HF because of its high per-processor memory demands in the absence of a garbage collector. In all graphs the lines for MCS-FG and HF-RC are coincident: we therefore present these results with suitably-labelled single lines.

Finally, we investigated the effects of varying the vector size. Table 3 shows μs per success on two 16-processor configurations. It is interesting to note the deleterious effect on performance caused by the increased contention occurring with smaller vector sizes, particularly for wider CASN. For example, on Power3-II the time per successful HF-RC CAS8 operation increases by 340% when the vector size reduces from 4096 to 256. With the larger vector 92% of operations succeed, but this drops to 52% for 256. The increased time per successful operation is due to the large amount of wasted work, but also to the heavy load placed on the machine's memory system as cache line ownership moves between CPUs.

Throughout all of our experiments we found that HF-RC outperforms the other non-blocking algorithm, IR, by a wide margin. Performance of HF-RC was closely comparable to that of the MCS-FG, the best lock-based algorithm. During these experiments we also recorded the ratio of minimum and maximum per-thread number of successful operations. Using this as a metric of fairness we found that HF-RC is at least as fair as MCS-FG.

7.3 Performance Bounds

We attempted to establish a best-case performance bound for CASN implementations on these systems. This was achieved by using a DUMMY function that performs a CAS1 on each of the N locations, but without any attempt to provide atomicity across the updates. For larger vector sizes (hence where contention is low) we found that the CPU time per operation used by DUMMY typically accounted for over 75% of that consumed by HF-RC. It is perhaps surprising that this simple operation takes such a large fraction of the time taken to complete the considerably more complex HF-RC and MCS-FG routines: For example, HF-RC requires three times as many CAS1 operations.

This discrepancy is because the cost of individual CAS1 operations vary considerably depending on whether the location's cache line is already held in an exclusive state in the local cache, or whether such a request must be issued to all other CPUs. When a CASN operation is started the locations that are to be updated are initially unlikely to be held locally since the vector is being actively shared with other CPUs. In contrast, the CAS1 operations that manipulate the CASN descriptor are likely to be local unless 'helping' has occurred.

We reason that any implementation of CASN will have to incur the cost of gaining exclusive ownership of the locations to be updated, and hence the performance of DUMMY provides a reasonable lower bound. From these results we conclude that substantial improvement on HF-RC is unlikely.

8 Conclusion

The results show that our algorithm achieves performance comparable with traditional blocking designs while maintaining the benefits of a non-blocking approach. By reserving only a small and constant amount of space (0 or 2 bits per location) we obtain key benefits over other non-blocking designs: those bits can often be held in storage that is otherwise unused in aligned pointer values, letting us build CASN from a single-word CAS1 operation and letting us use it on ordinary data structures. In contrast, Israeli and Rappoport's design requires per-processor reserved bits and further ownership information. Current processors provide a maximum of a 64-bit `CAS1` so if pointers are themselves 64-bits then this prevents the `IR` design from being used without substantial limitations on the data being held (or from being implementable at all much beyond the 32-processor results shown here). Moir's `LL/SC` constructions avoid per-processor reservations, but they are not amenable to use with natural pointer representations and their space reservations still grow with the level of concurrency [14]. Our work provides compelling evidence that `CAS1` is sufficient for practical implementations of concurrent non-blocking data structures, in contrast to other work that mandates `CAS2`.

References

1. S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.
2. J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proc. 14th PODC*, pp 184–193. Aug. 1995.
3. J. H. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects on priority-based systems. In *Proc. 16th PODC*, pp 229–238, Aug. 1997.
4. D. L. Detlefs, C. H. Flood, A. T. Garthwaite, P. A. Martin, N. N. Shavit, and G. L. Steele Jr. Even better DCAS-based concurrent deques. In *Proc. 14th DISC*, LNCS 1914, pp 59–73, Oct. 2000.
5. M. Greenwald. *Non-blocking synchronization and system design.* PhD thesis, Stanford University, Aug. 1999.
6. T. L. Harris. A pragmatic implementation of non-blocking linked lists. In *Proc. 15th DISC*, LNCS 2180, pp 300–314, Oct. 2001.
7. M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM TOPLAS*, 15(5):745–770, Nov. 1993.
8. M. Herlihy, V. Luchangco, and M. Moir The repeat offender problem: a mechanism for supporting dynamic-sized, lock-free data structures. In *Proc. 16th DISC*, 2002.
9. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
10. A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proc. 13th PODC*, pp 151–160, Aug. 1994.
11. H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, Department of Computer Science, June 1991.
12. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 9(1):21–65, Feb. 1991.
13. M. M. Michael Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proc. 21st PODC*, July 2002.
14. M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proc. 16th PODC*, pp 219–228, Aug. 1997.
15. M. Moir. Transparent support for wait-free transactions. In *Distributed Algorithms, 11th International Workshop*, LNCS 1320, pp 305–319, Sept. 1997.
16. N. N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th PODC*, pp 204–213, Aug. 1995.
17. J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th PODC* pp 214–222, Aug. 1995.