A Wait-free Multi-word Atomic (1,N) Register for Large-scale Data Sharing on Multi-core Machines

Mauro Ianni, Alessandro Pellegrini
DIAG–Sapienza Università di Roma, Italy
Email: {mianni,pellegrini}@dis.uniroma1.it

Francesco Quaglia
DICII–Università di Roma Tor Vergata, Italy
Email: francesco.quaglia@uniroma2.it

***Abstract*—We present a multi-word atomic (1,N) register for multi-core machines exploiting Read-Modify-Write (RMW) instructions to coordinate the writer and the readers in a wait-free manner. Our proposal, called Anonymous Readers Counting (ARC), enables large-scale data sharing by admitting up to $2^{32}-2$ concurrent readers on off-the-shelf 64-bit machines, as opposed to the most advanced RMW-based approach which is limited to 58 readers. Further, ARC avoids multiple copies of the register content while accessing it—this affects classical register's algorithms based on atomic read/write operations on single words. Thus, ARC allows for higher scalability with respect to the register size.**

I. Introduction

Hardware-based atomicity facilities offered by multi-core computing platforms to manage single-word shared-objects are not sufficient to automatically guarantee atomicity when concurrent threads manipulate multi-word objects. In this article we face such an issue by providing a pragmatic design and implementation of a shared-object algorithm in multi-processor/multi-core shared-memory machines. Specifically, we present Anonymous Readers Counting (ARC), which is an atomic (1,N)—one writer, N readers—register of arbitrary length. We emphasize that providing optimized (1,N) registers is a relevant objective since they constitute building blocks to realize more general (M,N) registers, as already shown by several works (see, e.g., [1]).

As the core property enabling scalability, ARC guarantees *wait-freedom* of both write and read operations. In fact, it uses no locking scheme, and guarantees that no operation fails and no retry-cycles are ever needed. This is achieved by relying on Read-Modify-Write (RMW) atomic instructions offered by conventional Instruction Set Architectures (ISAs), which are exploited to manipulate meta-data that are used by concurrent threads to coordinate themselves when performing register operations.

A close literature proposal based on RMW instructions, which still guarantees wait-freedom of read/write operations on (1,N) registers, is the one in [2]. However, this proposal allows up to 58 readers only on conventional 64-bit machines, while ARC can manage up to $(2^{32}-2)$ readers, thus enabling a huge scale-up in the level of concurrency. Also, the approach in [2] is based on deterministically forcing synchronization (via RMW instructions) upon any read operation, even in scenarios where the register's content has not been modified by the writer since the last read by the reader. ARC avoids executing RMW instructions in such situations, since it detects whether the last accessed snapshot of the register is still consistent by only relying on conventional memory-read machine instructions.

As opposed to more historical solutions for wait-free atomic (1,N) registers in shared-memory platforms [3], which only exploit atomic read/write operations (not RMW instructions) of individual memory words, we avoid multiple copies of the register content when performing either read or write operations. This allows for better scalability of ARC with respect to the size of the register content. Still related to buffer management, ARC adheres to the classical lower bound of $N+2$ buffers [4] keeping the different snapshots of the (1,N) register content, to be accessed in wait-free manner in some linearizable execution of read/write operations by the concurrent threads. Overall, compared to literature proposals, ARC enables definitely scaled up amounts of concurrent readers with no increased memory footprint and by not imposing extra memory-copy operations.

We also report experimental data collected on a 40 vCPUs virtual platform hosted by Amazon which show the advantages from ARC compared to literature solutions.

II. Anonymous Readers Counting

A. Basics

A *multi-word shared register* is an abstract data structure shared by concurrent processes[1] [3], [5]. Each process is allowed to perform two operations on the register: a *read*, which retrieves the most up-to-date value kept by the register, and a *write*, which stores a new register's value. We consider *asynchronous* processes. Also, the operations by a same process are assumed to execute sequentially.

The weakest class to which a register can belong is that of *safe* registers [4]. A register is safe if its correct value is guaranteed to be retrievable only if no concurrency is

[1]From now on we use the term 'process' and 'thread' interchangeably since the classical literature on register algorithms uses the term 'process' to indicate the active entity that can operate on the register.

allowed among reads and writes. Considering that we target concurrent objects, we consider a stronger class, namely *regular* registers. These are defined in terms of possible *execution histories* of concurrent read/write operations. In particular, each operation O on the register has a wall-clock time duration, which can be denoted as $[O_s, O_e]$ where O_s and O_e are the starting and ending instants, respectively. A regular register is one that is safe, and in which a read operation that overlaps (in time) a series of write operations obtains either the register value before the first of these writes or one of the values being written [4]. Introducing a reading function π to assign a write w on the register to each read r such that the value returned by r is the value written by w, and defining a precedence relation on the operations leading to a strict partial order '$\rightarrow$' [4], a regular register always respects the following property:

- **No-past**. There exists no read r and write w such that $\pi(r) \rightarrow w \rightarrow r$.

Nevertheless, in a regular register multiple reads executed concurrently to a write may not "agree" on the same value.

An even stronger class of registers is that of *atomic* registers [2]: *A shared register is atomic iff it is regular and the following condition holds for all possible executions:*

- **No New-Old inversion**. There exist no reads $r1$ and $r2$ such that $r1 \rightarrow r2$ and $\pi(r2) \rightarrow \pi(r1)$.

With an atomic register, reads can be separated among those "happening" before or after the linearization point [6], [7] of some write. This categorization marks the difference among the concurrent reads that can return the old value and those which need to return the new value. In particular, if two reads from the same process overlap a write then the later read cannot return the old value if the earlier read returns the new one. Atomic registers have been shown to be linearizable [8].

B. Memory Consistency Model

We assume the *Total Store Order* (TSO) [9] memory consistency model, which is typical of most off-the-shelf platforms (e.g. SPARC and x86). With TSO, CPU-cores use *store buffers* to hold the stores committed by the overlying pipeline until the underlying memory hierarchy is able to process them. A store leaves the buffer whenever the cache line to be written is in a coherence state such that the update can be safely performed. TSO allows *store bypasses*: even if a CPU-core outputs a write before a read, their order on memory (as seen by other CPU-cores) can be reversed.

Store bypasses can affect the correctness of synchronization algorithms for concurrent processes only relying on individual read/write operations (just like [5]). On the other hand, TSO-based architectures offer particular instructions, called *memory fences*, which enable recovering sequential consistency by explicitly flushing store buffers before executing any other memory operation, thus allowing to preserve the ordering across subsequent read/write operations.

For scenarios where process synchronization requires to atomically perform pairs (or more) operations, memory fences do not suffice. To cope with this issue, TSO-based conventional architectures offer *Read-Modify-Write* (RMW) instructions, whose execution directly interacts with cache controllers so as to ensure that cache lines keeping *synchronization variables* are held in an exclusive state until a couple of read/write operations are executed atomically [9]. In ARC we exploit the following RMW instructions: *atomic exchange*, which atomically reads the content of a memory location and updates its value; *add and fetch*, which increments a memory location and reads the updated value; *atomic inc*, which atomically increments the value of a memory location.

C. The Register Algorithm

In our algorithm we use $N+2$ buffers to keep different snapshots of the register value. Also, we exploit a single-word synchronization variable called `current`. It is a 64-bit shared variable divided into two fields: index, keeping the index of the slot containing the most up-to-date register value, and counter, namely the readers' presence counter (the number of standing concurrent reads on the slot targeted by index). index is 32-bit wide, so up to $(2^{32}-2)$ concurrent readers are allowed[2]. Additionally, our register data structure is made up by $N+2$ slots forming an array which we refer to as `register[]`. Each array slot is an instance of a data structure containing the following fields:

`r_start`	The number of read operations started on the slot since its last update.
`r_end`	The number of read operations completed on the slot since its last update.
`size`	The size of the register value stored in the slot.
`content`	A pointer to the memory location (the buffer) where the register content is stored.

The `size` field is introduced since we support writes (hence reads) of different sizes, meaning that each register value can have a different size (clearly up to some admissible maximum). Also, with no loss of generality, while presenting the register pseudo-code we assume that the buffer pointed by the `content` field of the register slot is already allocated, and that it can host the maximum sized register content (depending on the usage scenario). In any real implementation of our register algorithm, dynamic buffer allocation/release, with each buffer made up by the amount of bytes fitting the size of the register value to be stored upon write operations could be employed. The initial setup of the register data structure is shown in Algorithm 1.

[2]We have selected 32 as a meaningful value for common off-the-shelf architectures which use 64-bit words and RMW instructions targeting 64-bit memory locations. In different architectures, this could be set to an even larger value, by simply having the `current` variable enlarged in size, depending on the actual size of memory locations targeted by RMW instructions.

Algorithm 1 Register initialization.

```
1: procedure INIT($content$, $size$)
2:     for all $slot \in [0, N+1]$ do
3:         $register[slot].size \leftarrow 0$
4:         $register[slot].r\_start \leftarrow 0$
5:         $register[slot].r\_end \leftarrow 0$
6:     MEMCOPY($register[0].content, content, size$)
7:     $register[0].size \leftarrow size$
8:     $current \leftarrow N$                                    ▷ I1
```

Algorithm 2 The atomic register read operation.

```
1: procedure READ()
2:     $index \leftarrow current \gg 32$                          ▷ R1
3:     if $last\_index = index$ then
4:         $entry \leftarrow register[last\_index]$
5:         return $\langle entry.content, entry.size\rangle$        ▷ R2
6:     ATOMICINC($register[last\_index].r\_end$)                ▷ R3
7:     $tmp\_curr \leftarrow$ ATOMICADDANDFETCH($current, 1$)    ▷ R4
8:     $last\_index \leftarrow tmp\_curr \gg 32$                 ▷ R5
9:     $entry \leftarrow register[last\_index]$
10:    return $\langle entry.content, entry.size\rangle$
```

With no loss of generality, we assume that the register keeps its initial value into `register[0]`, and that the other $N+1$ entries are all available for posting some new register value.

Algorithm 2 shows the pseudo-code for the read operation. By exploiting the `AtomicAddAndFetch` instruction targeting `current`, a reader process is able to atomically retrieve the index of the slot containing the most up-to-date register value and increment the corresponding presence counter (**R4**). This allows us to enforce *visible reads* [10], although we do this in an *anonymous way*. In fact, the presence counter is not used to indicate who has started reading the up-to-date register value, rather how many processes did it. The index of the slot from which the up-to-date value is to be found is extracted by executing bitwise instructions on the value returned by `AtomicAddAndFetch`.

We consider a read operation from a slot as concluded when the reader reads again from the register. When, this happens, the `r_end` counter of the slot from which the reader took the register value upon its last read is incremented atomically. A special case occurs when the already-read slot still keeps the most up-to-date register value (**R2**). In this case, `r_end` is not incremented to indicate that the reader did not yet conclude its operations on the slot—a new read is just starting, bound to that same slot. Incrementing `r_end` only when moving to another slot allows us to avoid overflows of counter variables (**R3**). Thus, we enable an infinite number of reads (by any reader) to occur on a slot that still keeps the up-to-date register value. In order to remember from which slot the reader took the register value upon its last read we use the `last_index` variable, where we load the index of the target slot for the read operation each time the reader accesses a newer register value (**R5**). The check on whether the last accessed register value is still the most up-to-date is executed by loading the index

Algorithm 3 The atomic register write operation.

```
1: procedure WRITE($content$, $size$)
2:     pick $slot$ such that $slot \neq last\_slot \wedge register[slot].r\_start = register[slot].r\_end$   ▷ W1
3:     MEMCOPY($register[slot].content, content, size$)
4:     $register[slot].size \leftarrow size$
5:     $register[slot].r\_start \leftarrow 0$
6:     $register[slot].r\_end \leftarrow 0$
7:     $old\_curr \leftarrow$ ATOMICEXCHANGE($current$, $slot \ll 32$)   ▷ W2
8:     $old\_slot \leftarrow old\_curr \gg 32$
9:     $register[old\_slot].r\_start \leftarrow old\_curr \; \& \; (2^{32}-1)$   ▷ W3
10:    $last\_slot \leftarrow slot$
```

kept by `current` (**R1**) as soon as the read operation starts, and then comparing it with `last_index`. Given that the value of `current` is manipulated by any process (including the writer, as we will show) via RMW instructions only, then the index value returned by reading `current` (**R1**) is guaranteed to represent a correct snapshot of the shared synchronization variable we use in our register algorithm under the assumed TSO memory consistency model.

At startup `current` is initialized to N (**I1**). This sets its most-significant 32 bits (the index field) to zero and initializes the counter field as if all the readers had already started reading from the 0-th (initially-valid) slot. Therefore, if no update is ever made on the register's content, readers will indefinitely read this value (**R1**).

The pseudo-code for the write operation is shown in Algorithm 3. Upon writing, the writer process selects a free slot, namely a slot which is not currently bound to any not yet finalized read operation by whichever process, and which is different from the slot that was used for the last write operation (say the one kept by `current`). In compliance with the initialization of the register, we assume that the `last_slot` local variable kept by the writer, indicating the last slot used for a write, is initialized to the value 0. In fact, at init-time the initial register content is posted onto the 0-th slot. The writer detects if no more processes are currently reading from a slot by checking whether the two counters `r_start` and `r_end` associated with the slot keep the same value. The writer then performs a copy operation of the new value to the selected slot, and updates all the fields of the slot entry. In particular, it sets both `r_start` and `r_end` to zero, and `size` to the actual size of the new register value that is being stored. Then, by using an `AtomicExchange` instruction (**W2**), the writer changes the content of the `current` shared synchronization variable so as to "publish" the index of the new slot from which readers can start performing read operations. Given that the update of `current` is based on the execution of an RMW instruction, the content of the slot selected for the new write operation is guaranteed to be coherent when the `current` variable is updated under the assumed TSO memory consistency model. In other words, if a reader gets the updated `current` value (**R4**) and accesses the target slot, the accessed data are guaranteed to be coherent with the corresponding updates performed by the writer.

The new value of `current` which is atomically written by the writer (**W2**) has a counter field set to zero, telling that the new version has not yet been read by any process. The `AtomicExchange` allows to retrieve as well the old value of `current`, which is loaded into the `old_current` variable local to the writer. This is used by the writer to extract the old counter field, and store its value in the `r_start` field of the old (the last written) slot (**W3**). In this way, the number (not the identity) of readers which started an operation on the old slot is "freezed" into the slot management meta-data. We note that, after such freezing takes place for some slot, the corresponding values `r_start` and `r_end` are such that `r_start` $\geq$ `r_end`. But eventually these two values will be the same, which is the condition telling the writer that the slot has been released by all the readers since they moved to some fresher slot. In fact, the condition `r_start` $=$ `r_end` indicates to the writer that the slot is free again (**W1**). On the other hand, any written slot that is never accessed by any reader up to the point in time where some newer register value is atomically published by the writer, will have its `r_start` and `r_end` fields both set to zero, which implies it is a free slot.

The formal proof of correctness of ARC, in terms of atomicity and wait-freedom, can be found in [11].

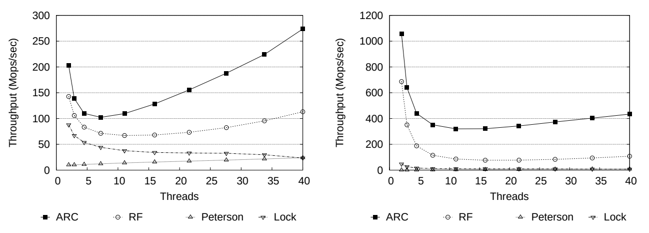
III. Experimental Results

We compared ARC with a few literature proposals, namely the Readers-Field (RF) wait-free algorithm presented in [2], still based on RMW instructions, and Peterson's wait-free algorithm [5]. Additionally, we included a classical lock-based approach not ensuring wait-freedom. All these algorithms have been implemented according to their original specification by relying on mixed C/ASM programming[3]. In all the implementations[4] we relied on pre-allocated buffers, and the "process entity", encapsulating the sequence of read/write register operations, is instantiated via an individual thread. We tested the algorithms on an Amazon m4.10xlarge instance equipped with 2.4 GHz Intel Xeon E5-2676 v3 (Haswell) processors offering a total capacity of 40 vCPUs, equipped with 160 GB of RAM. This virtual machine runs Ubuntu Server 14.04 LTS (kernel 4.2).

We recall again that ARC and RF not only differ by the different amounts of readers they can handle—58 in RF vs $(2^{32} - 2)$ in ARC. Rather, they also differ by the way RMW instructions are exploited along the execution path of read/write register operations. This makes a comparative analysis of the two algorithms interesting independently of the huge readers' count scale up admitted by ARC.

[3]For Peterson's algorithm ASM is used to nest memory-fence instructions that simulate the required sequential consistency memory model.

[4]Code available at https://github.com/HPDCS/ARC.



(a) 4KB register size (b) 128KB register size

Figure 1. Throughput with different register size values.

In our tests one thread continuously executes write operations on the register, while all the others continuously execute read operations on the register. Each reported sample is the average over 10 runs, with each run made up by at least 2×10^6 read/write operations. Also, the different plots refer to 2 different sizes of the register, a minimal size of 4KB (a single operating system page), and a large size of 128KB. By the plots in Figure 1 we see how both ARC and RF outperform the other solutions at any thread count. Also, ARC outperforms RF at any thread count and for any tested register size, providing up to an order of magnitude better throughput for larger size. The reason for this behavior is that RF executes an RMW instruction (i.e. a `FetchAndOr`) upon any read, while ARC executes an RMW instruction only if the write operation of a newer register value is serialized before the execution of the statement **R1** of the read operation in Algorithm 2. Hence, ARC is more efficient (since it avoids the execution of RMW instructions) upon reading a register content that is still valid (i.e. it did not change since the last read operation executed by the same thread). This scenario shows up when increasing the level of concurrency of read operations or when the write operation takes longer time due to the larger size of the register content to be posted by the writer—we recall that a memory copy is executed upon a write. In both cases more threads will likely find a not-yet-updated register value upon subsequent reads, a scenario which is captured more efficiently by ARC, compared to RF, just avoiding the execution of RMW instructions.

IV. Conclusions

We have presented Anonymous Readers Counting, a multi-word wait-free atomic (1,N) register algorithm targeting shared-memory TSO-consistent parallel architectures. It enables up to $(2^{32} - 2)$ readers on 64-bit machines and avoids any intermediate copy of the register content upon any operation, while still using the classical lower bound of $N + 2$ buffers for ensuring wait-freedom. It exploits Read-Modify-Write (RMW) instructions by also reducing the impact of actually running these instructions compared to the reference literature proposal in [2], which also has the disadvantage of handling up to 58 readers only.

References

[1] M. Li, J. Tromp, and P. M. B. Vitányi, "How to share concurrent wait-free variables," *Journal of the ACM*, vol. 43, pp. 723–746, 1996.

[2] A. Larsson, A. Gidenstam, P. H. Ha, M. Papatriantafilou, and P. Tsigas, "Multiword atomic read/write registers on multiprocessor systems," *Journal of Experimental Algorithmics*, vol. 13, no. 1, p. 1.7, 2009.

[3] L. Lamport, "Concurrent reading and writing," *Communications of the ACM*, vol. 20, no. 11, pp. 806–811, 1977.

[4] L. Lamport, "On interprocess communication," *Distributed Computing*, vol. 1, pp. 86–101, 1986.

[5] G. L. Peterson, "Concurrent Reading While Writing," *ACM Transactions on Programming Languages and Systems*, vol. 5, pp. 46–55, 1983.

[6] M. P. Herlihy and J. M. Wing, "Axioms for concurrent objects," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 13–26, 1987.

[7] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[8] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[9] D. J. Sorin, M. D. Hill, and D. A. Wood, "A Primer on Memory Consistency and Cache Coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[10] J. Burns and N. A. Lynch, "Mutual Exclusion Using Invisible Reads and Writes," in *Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*, pp. 833–842, 1980.

[11] M. Ianni, A. Pellegrini, and F. Quaglia, "A wait-free multi-word atomic (1,N) register for large-scale data sharing on multi-core machines," *CoRR*, vol. cs.DC/1707.07478, 2017.