Anonymous Readers Counting: A Wait-free Multi-word Atomic Register Algorithm for Scalable Data Sharing on Multi-core Machines

Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia

Abstract—In this article we present Anonymous Readers Counting (ARC), a multi-word atomic (1,N) register algorithm for multi-core machines. ARC exploits Read-Modify-Write (RMW) instructions to coordinate the writer and reader threads in a wait-free manner and 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 on the same kind of machines. Further, ARC avoids multiple copies of the register content when accessing it—this is a problem that affects classical register algorithms based on atomic read/write operations on single words. Thus it allows for higher scalability with respect to the register size. Moreover, ARC explicitly reduces the overall power consumption, via a proper limitation of RMW instructions in case of read operations re-accessing a still-valid snapshot of the register content, and by showing constant time for read operations and amortized constant time for write operations. Our proposal has therefore a strong focus on real-world off-the-shelf architectures, allowing us to capture properties which benefit both performance and power consumption. A proof of correctness of our register algorithm is also provided, together with experimental data for a comparison with literature proposals. Beyond assessing ARC on physical platforms, we carry out as well an experimentation on virtualized infrastructures, which shows the resilience of wait-free synchronization as provided by ARC with respect to CPU-steal times, proper of modern paradigms such as cloud computing. Finally, we discuss how to extend ARC for scenarios with multiple writers and multiple readers—the so called (M,N) register. This is achieved not by changing the operations (and their wait-free nature) executed along the critical path of the threads, rather only changing the ratio between the number of buffers keeping the register snapshots and the number of threads to coordinate, as well as the number of bits used for counting readers within a 64-bit mask accessed via RMW instructions—just depending on the target balance between the number of readers and the number of writers to be supported.

Index Terms—Atomic registers, Shared-memory, Multi-core computing, Wait-free synchronization, Instruction-Set-Architecture.

1 Introduction

Hardware-based atomicity facilities offered by multi-core computing platforms for managing single-word shared-objects are not sufficient to automatically guarantee atomicity when concurrent threads manipulate multi-word objects. Synchronization algorithms are therefore needed to enable atomic read/write operations on this type of objects. Also, the extreme level of scale-up of modern computing platforms, with projection towards exascale computing, demands for shared-object management algorithms that are capable of efficiently supporting huge levels of concurrency.

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 (i.e., made up by an arbitrary number of words, which can change over time, possibly upon each update of the register). ARC exhibits the following capabilities:

- it is devised for a huge scale-up of the number of concurrent threads to be managed;
- it targets the optimization of the actual execution path of the threads along multiple dimensions: locality, time complexity and actual cost of machine instructions to be executed.

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]). We also provide one such extension in this article, showing how ARC can be trivially adapted to the (M,N) case, essentially with no change of the tasks executed along the critical path of read/write operations as compared to the (1,N) scenario.

As its core property enabling scalability, ARC guarantees *wait-freedom* [2] of both write and read operations. Indeed, 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) 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. Moreover, ARC does not require strong memory consistency support in the underlying hardware,

- *M. Ianni and A. Pellegrini are with the Department of Computer, Control, and Management Engineering, Sapienza University of Rome. E-mail: {mianni,pellegrini}@dis.uniroma1.it*
- *F.Quaglia is with Dipartimento di Ingegneria Civile e Ingegneria Informatica, Università di Roma "Tor Vergata". E-mail: francesco.quaglia@uniroma2.it*

Manuscript received N/A; revised N/A.

such as *Sequential Consistency*. Indeed, it is suited for off-the-shelf processors—such as x86 processors—offering the weaker *Total Store Order* consistency model. This makes ARC employable in a wide variety of hardware platforms.

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 [3]. However, this proposal allows a maximum of 58 readers 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 [3] deterministically forces 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 (it is the most up to date one within the linearizable history of read/write accesses) by only relying on conventional memory-read instructions.

The benefits by this optimization on performance and energy efficiency are non-minimal, as we show experimentally, given the effects of RMW instructions on the interconnection among CPU-cores. For example, modern Intel-based architectures relying the QuickPath Interconnect [4] require message passing among CPU-cores when executing RMW instructions. Furthermore, these effects can be amplified when a memory location updated by a RMW instruction is split across different cache lines, as shown in [5].

As opposed to more historical solutions for wait-free atomic (1,N) registers in shared-memory platforms [6], which only exploit atomic read/write operations 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. Also, ARC adheres to the classical lower bound of $N + 2$ buffers [7] 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, thus favoring locality.

Furthermore, ARC allows constant time for read operations, jointly guaranteeing amortized constant-time for write operations. This is not guaranteed by the RMW-based approach in [3], since it requires $O(N)$ time for write operations—an aspect that is clearly related to the reduced amount of readers admitted by such register algorithm.

Beyond presenting ARC, we also provide a proof of its correctness. Further, we report experimental data showing the benefits from our proposal compared to a few literature solutions. Performance data have been collected on a physical machine equipped with 48 CPU-cores and on a virtual platform hosted by Amazon equipped with 40 vCPUs. As a last note, our experimental evaluation has been based on user-space code implementing ARC, but nothing prevents ARC to be integrated within lower-level software layers, such as an operating system kernel.

The remainder of this article is organized as follows. In Section 2 we discuss related work. ARC is presented in Section 3. Its correctness proof is provided in Section 4. The variation of ARC coping with multiple writers is provided in Section 5. Experimental results are reported in Section 6.

2 Related Work

We target shared-objects in multi-processor/multi-core machines, to be managed in a wait-free manner. According to [2], wait-freedom allows any concurrent operation on the shared-object to execute in a finite number of steps, regardless of any action carried out by other concurrent operations. This is not guaranteed neither by classical lock-based synchronization schemes [8] nor by lock-free ones [9], [10]. Wait-freedom appears as a mandatory means to efficiently handle concurrent operations on shared-objects in systems with large/huge amounts of concurrent threads.

A (1,N) register algorithm for multi-processors has been provided by Lamport in [6]. This solution enables wait-free writes, but only guarantees lock-free read operations, since the writer can force slow-running readers to retry their read operations indefinitely. A fully wait-free solution has been presented by Peterson [11], which marked the begin of a long running research path towards the construction of wait-free solutions to the readers/writers problem. Along this path we find proposals dealing with (1,1) [7], [12], (1,N) [11], [13], and (M,N) registers [11], [14]. A common aspect that characterizes these proposals is that they build wait-free multi-word registers by relying on single-word read/write registers, just based on atomic single-word read/write instructions. Thus, they do not exploit synchronization facilities offered by conventional multi-processor/multi-core machines, such as RMW instructions like Compare-and-Swap (CAS). The disadvantage lies in that, in order to assess the validity of a multi-word atomic read/write operation, it must be carried out multiple times (e.g., 2 times in [11]), which may impair performance (as well as energy efficiency) especially when scaling up the size of the register. In our approach we avoid this drawback by avoiding at all multiple copies of the register content upon both read and write operations. In particular, we support write operations with a single copy of the new register content into the target buffer. Also, read operations do not need any intermediate data copy, since the reading process can directly read data from the buffer originally targeted by the write operation that is serialized before the read itself. Hence, in ARC, accessing the register in read mode only entails retrieving the correct buffer address.

Several proposals [15], [16], [17], [18] allow to realize a wait-free register by relying on a wait-free universal construct [19]. This is a design choice that we have explicitly avoided, making our proposal mostly orthogonal. In fact, the employment of a universal construct does not allow capturing the intrinsic properties of the different register operations (read vs write). In turn, this might reduce performance since the number of synchronization steps might be much larger than what strictly required (just depending on the different nature of the operations). As an example, the work in [18] realizes a read operation as a generic one, making it at least as heavyweight as a write operation, while in ARC we have explicitly differentiated the implementations of read

and write operations, so as to jointly optimize their execution path. Moreover, a number of synchronization steps not adhering to the required minimum might have a negative impact on both scalability and energy efficiency also because of the effects on the underlying memory hierarchy.

Another difference with ARC is that the work in [18] requires $O(N^2)$ buffers for achieving wait freedom, while we stick to the traditional lower bound of $N + 2$ buffers, and slide towards quadratic memory only for the case of multiple writers. Quadratic memory cost is also paid by the proposal in [17], together with $O(N)$ time due to the reliance on hazard pointers. ARC shows linear time only in some corner cases of write operations, since it provides constant-time for reads and amortized constant-time for writes.

Among the aforementioned works exploiting the concept of universal constructor, [16] is the only one using RMW instructions. Nevertheless, wait-freedom is guaranteed by having all threads record the operation that they want to do—either a read or a write—in a shared buffer. Then, all the threads attempt at the same time to complete all the registered operations, ensuring that only one of them actually succeeds. This implies a total of $O(N^2)$ attempts to carry out N operations. On the other hand, we keep the wait-free nature of the algorithm, while avoiding that multiple threads carry out the same operations.

The interest in exploiting increasingly scalable synchronization approaches while managing shared-objects has recently grown also by the side of operating system software. Along this path we find the Read-Copy-Update (RCU) mechanism supported by the Linux kernel [20]. This mechanism allows readers not to block and to observe consistent states of a shared data structure even though updates are in progress. However, the mechanism is not actually wait-free since writers experience so called wait-for-readers periods, which are needed in order to detect whether readers may still require old and new copies of the data structure to be still in place for correct finalization of their read operations. Also, multiple writers need to synchronize in a critical section. Although the impact of writers' synchronization can be reduced by approaches like Read-Log-Update (RLU) [21], where facilities like Transactional Memory (TM) are proposed for this kind of synchronization, wait-freedom is not actually guaranteed. Our solution is instead fully wait-free, thus not suffering from blocking (or retry) phases. On the other hand, the approach in [21] targets usability of the synchronization scheme with different data structures, while we focus on the register abstraction.

To the best of our knowledge, the only literature proposal based on RMW instructions to support an atomic wait-free (1,N) register is the one in [3]. Here the authors use 64-bit atomic memory operations to update/retrieve a bit-mask indicating what is the buffer instance containing the updated version of the register content and which threads are reading this content version. The overall number of slots to be managed is $N+2$, as a classical minimum requirement for a wait-free (1,N) register. Hence, by partitioning the 64-bit mask into the two aforementioned portions—one for the buffer instance and the other for standing reads identification—the maximum number of admitted concurrent readers is 58. Compared to this approach, we use RMW instructions on 64-bit words in a completely different manner, since we do not associate individual bits with threads to indicate whether a given thread has a standing read on a given buffer instance. Rather, we adopt an anonymous scheme where registering a thread as a reader on a given buffer instance (a register snapshot) only entails incrementing a per-instance counter of standing reads—hence the name ARC for our proposal. As a consequence, we can host up to $2^{32} - 2$ concurrent readers, which is done by still relying on $N + 2$ buffers to keep the register content[1]. Overall, compared to the work in [3], our proposal handles scenarios with a large/huge increase of the amount of threads allowed to concurrently perform read operations. Hence, we enable scaled-up wait-free concurrency on the atomic register up to a level fitting the requirements of massively parallel applications hosted by huge (virtualized) parallel platforms. Also, the actual number of RMW instructions executed in our register algorithm under diverse workloads is typically lower than that of the algorithm in [3]. As we will show via experimental data, this leads to a reduced impact of RMW instructions on performance by our proposal.

3 Anonymous Readers Counting

3.1 Basics

A *multi-word shared register* is an abstract data structure that is shared by a number of concurrent processes[2] [6], [11]. 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 value. We consider *asynchronous* processes, meaning that no assumption is made on their relative speed or on the interleave of their operations. The operations by a same process are assumed to execute sequentially.

The weakest class to which a register can belong is the one of *safe* registers [7]. A register is safe if its correct value is guaranteed to be retrievable only if no concurrency is allowed (or happens) among reads and writes. Considering that we target concurrent objects, we consider a stronger class, namely *regular* registers.

Regular registers 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 [7]. 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 '$\rightarrow$' [7], a regular register always respects the following property:

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

1. A slightly different bits-to-counters association scheme is applied for the case of multiple writers—the (M,N) atomic register case.

2. From now on we use the terms '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.

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

By the *linearizability* property [10], [22], we can always find a *linearization point* which provides the illusion that each operation O takes effect instantaneously at some point between O_s and O_e. Consequently, a stronger class of registers is the one of *atomic* registers, defined according to the following criterion [3]:

Criterion 1. *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 π(r2) $\rightarrow$ π(r1).*

With an atomic register, reads can be separated among those "happening" before and after the linearization point 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. If two non-concurrent reads 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 [23].

3.2 Memory Consistency Model

Multi-processor/multi-core shared-memory systems offer *memory consistency models* [24] as kind of "contracts" among software developers and hardware manufacturers. They discriminate what software can expect to be guaranteed by the underlying hardware. A variety of consistency models exist, which are often presented as a set of rules.

The simplest memory consistency model is *sequential consistency*. In this model "the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [25]. This model ensures that all read and write instructions executed by any processor are observed in the same order by all the processors in the system. Peterson's algorithm [11] and several lock-based algorithms [9], [10] require sequential consistency.

We assume a weaker consistency model, namely *Total Store Order* (TSO) [24], which is used by most off-the-shelf platforms, such as x86, thus making our solution of general applicability. With TSO, CPU-cores usually use *store buffers* to hold the stores committed by the overlying pipeline until the underlying memory hierarchy is able to process them. In particular, 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 what is called *store bypass*: even if a CPU-core outputs a write before a read, their order on memory (as seen by other CPU-cores) can be reversed.

While TSO produces no damage in many applications—rather, it can provide a significant speedup due to a reduced latency on the memory hierarchy—synchronization based on shared-memory data must explicitly cope with this scenario. In fact, store bypasses can affect the correctness of synchronization algorithms (e.g. register algorithms) for concurrent processes only relying on individual read/write operations (just like [11]). On the other hand, TSO-based architectures offer particular instructions in their ISA, referred to as *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.

However, for scenarios where synchronization among processes requires to atomically perform pairs of operations (or more), memory fences do not suffice. To cope with this issue, TSO-based 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 [24]. This means that no other cache can keep the same line in read mode until the couple of operations completes.

Classical RMW instructions, which we exploit in ARC, are: *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.

3.3 The Register Algorithm

ARC uses $N + 2$ buffers to keep different snapshots of the register value, as produced along time by write operations. This number has already been proven to be the lower bound for achieving wait-freedom in (1,N) registers [7]. Having such a number of buffers allows each reader to keep a buffer for reading—possibly different across the N readers—while at least 2 buffers are still available to keep some up-to-date register value (the one written while the readers were concurrently reading the register) and the work-in-progress copy being produced by the writer, if any. We will refer to each of the $N + 2$ buffers as a slot of the register.

The core data structure we exploit in ARC is a single-word shared 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. The index field is 32 bits wide, therefore up to $2^{32} - 2$ concurrent readers are allowed[3].

Additionally, our register data structure is made up of $N + 2$ meta-data entries forming an array which we refer to as `register[]`. Each entry of this array is associated with a register slot. Also, the entry is an instance of a data structure containing the following four fields:

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

The `size` field is introduced since we support writes and reads of different sizes, meaning that each register value

3. 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 (or future) 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 ∈ [0, N + 1] do
3:          register[slot].size ← 0
4:          register[slot].r_start ← 0
5:          register[slot].r_end ← 0
6:      MEMCOPY(register[0].content, content, size)
7:      register[0].size ← size
8:      current ← N                                   ▷ I1
```

Algorithm 2 The atomic register read operation.

```
1:  procedure READ()
2:      index ← current ≫ 32                          ▷ R1
3:      if last_index = index then
4:          entry ← register[last_index]
5:          return ⟨entry.content, entry.size⟩        ▷ R2
6:      ATOMICINC(register[last_index].r_end)         ▷ R3
7:      tmp_curr ← ATOMICADDANDFETCH(current, 1)      ▷ R4
8:      last_index ← tmp_curr ≫ 32                    ▷ R5
9:      entry ← register[last_index]
10:     return ⟨entry.content, entry.size⟩
```

can have a different size. 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. Also, given that the memory allocation operation by the writer can be kept out of the critical path of the actual algorithmic steps of ARC, wait-freedom of read/write operations is still preserved by ARC even if the memory allocation system can lead to block the writer for buffer acquisition[4].

The initial setup of the register data structure is shown in Algorithm 1. With no loss of generality, we assume that the register is initialized to keep 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 (statement **R4**). This allows us to enforce *visible reads* [26], 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 where 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 as soon as the reader tries to read 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 (statement **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 (upon a subsequent read that finds a newer register value) allows us to avoid overflows of counter variables (statement **R3**). Thus, we enable an infinite number of reads 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 (which is local to a reader), where we load the index of the target slot for the read operation each time the reader accesses a newer register value (statement **R5**). The check on whether the last accessed register value is still the most up-to-date is executed by loading the index kept by `current` (statement **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` (statement **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 (statement **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[5]. Therefore, if no update is ever made to the register's content, readers will indefinitely read this value (statement **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 (namely, 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 initialization time the initial register content is posted to the 0-th slot. The writer detects if no other process is 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 (statement **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

4. Essentially, the problem of allocating memory in a wait-free manner is fully orthogonal to the problem of guaranteeing wait-freedom of read/write operations on the allocated memory buffers according to the atomic register rules.

5. With no loss of generality our algorithmic notation is assuming a little-endian 64-bit processor, like x86-64. However, it can be easily adapted to big-endian processors.

Algorithm 3 The atomic register write operation.

```
1:  procedure WRITE(content, size)
2:      pick slot such that slot ≠ last_slot ∧ register[slot].r_start = register[slot].r_end   ▷ W1
3:      MEMCOPY(register[slot].content, content, size)
4:      register[slot].size ← size
5:      register[slot].r_start ← 0
6:      register[slot].r_end ← 0
7:      old_curr ← ATOMICEXCHANGE(current, slot ≪ 32)                                          ▷ W2
8:      old_slot ← old_curr ≫ 32
9:      register[old_slot].r_start ← old_curr & (2^32 − 1)                                     ▷ W3
10:     last_slot ← slot
```

memory consistency model. In other words, if a reader gets the updated `current` value (statement **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 (statement **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 (statement **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`. Eventually these two values will be the same, which is the condition telling the writer that the slot has been released by all 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 (statement **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 available for a new write.

3.4 Speeding-up Free Slot Searches

By the pseudo-code of ARC, read operations can be trivially shown to take constant-time. On the other hand, write operations require searching for a free slot among $N + 2$ (statement **W1**), which would imply linear time complexity. To provide amortized constant time for write operations, in particular for the slot-search operation, readers that complete their read from a slot by incrementing the corresponding `r_end` counter (i.e. they release the slot), can check whether this counter is equal to the `r_start` counter associated with the same slot. If this is true, then by the register algorithm structure it means that the slot can be reused for subsequent writes. Hence, a reader detecting such an equality can post into another shared variable the index of the just-released slot. This can be used by the writer as a *proposal* to start searching for a free slot. This proposal will always correspond to an actually-free slot (hence enabling constant time retrieval of the free slot upon write operations) except for the corner case where the writer already took the same slot for some already-issued write having observed its release before the reader posted its proposal.

4 CORRECTNESS PROOF

By code construction, all invocations to READ() are guaranteed to complete in a finite number of steps. Hence reads are guaranteed to be wait-free. As for the WRITE() operation, completion within a finite number of steps is guaranteed if the free-slot search operation carried out at the beginning of the write operation (statement **W1**) completes in a finite number of steps. This is true if it is guaranteed that at least one slot different from the last one used for a register write is in a stable state such that its `r_start` and `r_end` fields are equal. This is proven in the following lemma:

Lemma 4.1. *Upon starting a write operation at least one of the $N + 2$ register slots, which is different from* `last_slot`*, is such that* `r_start` *and* `r_end` *keep the same value. Also, for this slot, these values do not change while the writer executes statement* **W1**.

Proof. This proof is based on two disjoint cases analysis:
Case 1. The writer performs its first write on the register. In this case, all the `r_start` and `r_end` fields are still found to be set to the value 0. This is because no reader could have updated any `r_end` field in any slot since this can only happen if a newer register value is found upon a read operation, which is not the case since the writer did not yet post any new value, say `current` has never been updated. In fact, for a reader to update the `r_end` field of any slot, it necessarily needs to find the predicate in line 2 of the READ() operation not satisfied, thus sliding to the execution of statement **R4**. However, when running the statement **W1** upon its first write operation, the writer did not yet update the index filed of the `current` variable, so that no reader can have found the predicate in line 2 of the READ() operation unsatisfied. Also, no `r_start` field in any slot can ever change while the writer executes the **W1** statement during its first write operation, since this change is allowed to occur only at statement **W3** of the write operation, and this statement does not precede statement **W1**.

Overall, given that `last_slot` is set to 0 upon register initialization, all the $N + 1$ slots different from the 0-th one are such that their `r_start` and `r_end` fields are set to zero and cannot change while the writer executes the **W1** statement during its first write operation, at least one slot different from `last_slot` is such that its `r_start` and `r_end` fields keep the same value which will not change while the writer executes **W1**. Thus the claim follows.
Case 2. The writer performs the i-th write on the register. In this case, all the writes up to the $(i-1)$-th one have updated `current`, and the readers might have fetched the various values of `current`, also releasing a presence count unit each

time this happened. By the READ() operation pseudo-code, a reader leaves a presence count unit on some slot (updating the counter field of the variable `current`) only after having released a count unit on the `r_end` field of some other slot. Hence, at the time of executing the **W1** statement of the write operation, for all the `r_start` units freezed by the writer into the slots' meta-data upon performing writes (through statement **W3**) up to the $(i-1)$-th we have that:

$$\sum_{j=0}^{N+1} (register[j].r\_start - register[j].r\_end) \leq N \quad (1)$$

Hence, given that `r_start` and `r_end` fields are non-negative values, at the time of executing statement **W1** during the i-th write by the writer, for at least 2 different slots of the $N+2$ slots of the register, these same fields must have the same value.

Let us now prove that these values do not change while the writer executes statement **W1**. The `r_start` value of any slot can only be modified by the writer at statement **W4** of the WRITE() operation, thus it cannot change while the writer executes statement **W1**. Also, for any generic slot, the reader releases a presence count in the `r_end` field only after moving to another slot, and having released a presence count on the count field of that slot. Hence given that at the time of executing statement **W1** during the i-th write by the writer all the `r_start` values of the slots that have been written by the writer up to the $(i-1)$-th write, and possibly accessed by the readers, are already flushed into the slots' meta-data (see statement **W3** of the WRITE() operation), then no reader can update the corresponding `r_end` fields to a value greater than the corresponding `r_start` fields. So these updates cannot occur while the **W1** statement is in progress. Therefore, for the slots for which `r_start` and `r_end` are found to be equal upon starting statement **W1** at the writer, they are not allowed to be changed during the statement execution. Hence at least one slot which is different from `last_slot` is such that its `r_start` and `r_end` fields stably keep the same value while the writer executes statement **W1**. Thus the claim follows. □

We now prove consistency of concurrent read/write operations in ARC:

Lemma 4.2. *While the writer is executing a write operation on a slot, no reader will read the same slot until the write completes.*

Proof. Read operations bound to the initial snapshot of the register trivially satisfy the claim, since that snapshot is not written by the writer. Let us therefore focus on reads of the register snapshots that are different from those bound to the register initialization value. By the READ() operation pseudo-code, a read operation is always bound to the slot index that is returned at some point in time by atomically executing `AtomicAddAndFetch` on the `current` variable (see statement **R4** of the READ() operation). This is true also when subsequent reads by a reader process take an unchanged register content from a same slot, since the first of these reads must have necessarily executed the `AtomicAddAndFetch` instruction on `current` to retrieve the index of that slot. On the other hand, the `r_end` field of some slot is incremented by the reader only after moving to some new slot upon read operations. In fact, the **R3** statement of the READ() operation is executed only if the predicate in line 3 of that same operation is not satisfied.

Given that (i) the writer selects a slot x for writing only when it finds its `r_start` and `r_end` fields set to the same value, (ii) `r_start` is freezed into the meta-data of slot x only after it is no longer the current one, (iii) whichever slot x becomes again readable after its index is published into the `current` shared variable, (iv) TSO memory consistency guarantees that when the update of `current` is performed by the writer at statement **W2** of the WRITE() operation, so as to point to the x-th slot, all the data associated with the slot have already been flushed to memory, we have that any read will always observe a stable snapshot of the register when reading from the generic x-th slot. Hence the claim follows. □

We now prove regularity and atomicity of ARC:

Theorem 4.3 (Regular Register)**.** *Any read operation returns either the last written value, or one being concurrently written.*

Proof. By the structure of Algorithm 3 implementing the WRITE() operation, the update of `current` performed at statement **W2** represents the atomic memory operation that defines the linearization point for any write. If the write is linearized before the execution of statement **R1** of the READ() operation by some reader, a read always returns the last written value, say the one posted by the last write serialized before the read, since the serialization point of the read is determined by statement **R4**, which targets the same shared synchronization variable `current` whose atomic updates represent the serialization points of writes. Otherwise, if the statement **R1** of the READ() operation is executed before the update of the `current` shared synchronization variable by the write operation at statement **W2**, the read is correctly allowed to return the register value that was already stored. Hence the claim follows. □

Theorem 4.4. *Given two read operations r_1 and r_2 such that $r_1 \rightarrow r_2$, r_2 never returns a value older than the one returned by r_1.*

Proof. (By contradiction) By the proof of Theorem 4.3, a read executed before the linearization point of a write returns the old value (with respect to the execution of the write). Let us assume by contradiction that, given two reads r_1 and r_2 such that $r_1 \rightarrow r_2$, r_2 returns a value older than the one returned by r_1. Yet, the `current` synchronization variable is updated at statement **W2** of the WRITE() operation whenever the index of the most up-to-date slot changes. Therefore, for r_2 to read a value older than r_1, it has to read `current` before r_1. But this violates the precedence $r_1 \rightarrow r_2$. Hence the assumption is contradicted and the claim follows. □

Atomicity of our register algorithm trivially follows from Theorem 4.3 and Theorem 4.4 in combination.

5 Extension to the Multiple Writers Case

The extension of ARC to manage multiple writers is immediate, under the scenario where we admit that each of the M writers is allowed to use $N+2$ buffers for posting

updated register values. In fact, in this scenario, each of the M writers is allowed to retrieve a free slot for posting a new register value in a finite number of steps, independently of what register versions are currently being accessed by the N readers. For the (M, N) case, the index field of the `current` synchronization variable can keep track of any of the $M \times (N+2)$ buffers, and a writer publishing a new register value simply posts the same bit-mask used in the single writer case onto `current`, still indicating that initially zero readers are bound to the register value being posted. On the other hand, posting the new information on `current` via the *atomic exchange* instruction allows the generic writer to retrieve the index of the slot keeping the last posted register value, thus allowing to flush the current readers' count on the corresponding meta-data even if such slot was in charge of another writer. Clearly, the only limitation of this extension, in terms of actual concurrency levels that can be managed, is that we can no longer support $2^{32}-2$ readers, since more bits in the `current` synchronization variable needs to be used for keeping track of the index of any of the possible $M \times (N+2)$ slots used to keep the register values produced by the M writers. In particular, using h bits to keep track of all the possible slots in the index field of the synchronization variable `current` and k bits for the count field, the constraints that need to be satisfied, based on a 64-bit representation of `current`, are the following ones:

$$\begin{aligned} 2^h &= M \times (N+2) \\ 2^k &= N \\ k + h &\leq 64 \end{aligned} \tag{2}$$

In the following table we list a few possible solutions of the above equations, in terms of values of M and N which lead to respect all the expressed constraints:

M (admitted writers)	**N** (admitted readers)
2	2^{31}
2^{11}	2^{26}
2^{23}	2^{20}
2^{31}	2^{16}

By these data we see how ARC still allows for extremely high concurrency of both writers and readers in the multiple writers scenario, e.g. by admitting the order of one million writers and one million readers all together.

We additionally note that the pseudo-code presented for the case of single writer is still fully valid, with the only need to modify the value used for shifting the `current` synchronization variable (or other used bit-masks) in order to correctly manipulate (retrieve and update) its fields. Also, the search for a free slot operation executed by the writer (see line 2 of Algorithm 3) needs to be carried out exclusively among the slots bound to it, which are disjoint with respect to the slots bound to others. Finally, the mechanism of speeding up writes by making such a search by the writer executed in amortized constant time can still be put in place by simply keeping multiple hint-for-search variables each associated with a different writer (e.g. via hash association with the slots managed by that writer), each of which is updated by a reader that eventually releases a buffer bound to that writer—recall that such a reader is unique, since it corresponds to the one that lets the `r_end` variable associated with the slot acquire the same value as `r_start`.

6 Experimental Results

In this section we experimentally compare ARC with the Readers-Field (RF) wait-free algorithm presented in [3], still based on RMW instructions, with Peterson's wait-free algorithm [11] and with a classical lock-based approach (using read/write spin-locks) not ensuring wait-freedom. We decided to focus on the (1,N) configuration in our experiments, which is the most studied one in terms of actual optimization of the execution path of the threads. In any case we recall again that the execution path of ARC in the (M,N) configuration is essentially identical to the one of the (1,N) configuration.

All the compared algorithms have been implemented according to their specification by relying on the C programming language and Posix, plus the nesting of either RMW machine instructions used to manipulate synchronization variables (like in ARC and RF) or memory-fence instructions to guarantee correctness under TSO (like for Peterson's algorithm). Also, in all the implementations we relied on `mmap()` to pre-allocate all the buffers requested by each algorithm[6]. In all the implementations, the "process entity", encapsulating the sequence of read or write operations accessing the register, is instantiated via an individual thread.

We tested the different algorithms deploying their implementations on two different computing platforms, a physical one and a virtualized one. The former is equipped with four 1.9 GHz AMD Opteron 6168 processors and 128 GB of RAM. Each processor has 12 cores, for a total of 48 CPU-cores. The operating system is 64-bit Debian, with Linux Kernel 4.15. The virtualized platform is 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 as the operating system, with Linux Kernel 4.2.

We have conducted two different sets of experiments. In the first set, we generated a workload which is similar in spirit to the well-known *Hold-Model* [27]. With this workload, all the concurrent threads repeatedly execute kind of "dummy" operations on the register data structure—each write operation simply copies the same content to the register, and a read operation only retrieves the pointer to the register value. This is an extreme scenario in which data processing has zero latency, and threads make no other work than accessing the register data structure. The effect of this behavior is that the logical contention on the register data structure is maximal. Then we have associated read and write operations with actual processing—a write actually generates some data, and a read scans the register content. In this second scenario we can study the effect of different operations' latencies.

Before discussing performance results, 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 in the (1,N) configuration. Rather, they also differ by the way

6. The source code for all the tested implementations is available at `https://github.com/HPDCS/ARC`.

RMW instructions are exploited along the execution path of read/write operations accessing the register. This aspect makes a comparative analysis of these two algorithms interesting independently of the huge scale up of the level of concurrency admitted by ARC.

In Figure 1 we report throughput values (read/write operations per time unit) while varying the number of threads for deploys of the different register implementations on the 48 CPU-core physical machine. In these tests, one thread continuously executes write operations on the register, while all the others continuously execute read operations. 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 4 different sizes of the register, a minimal size of 128B, a small size of 4KB, an intermediate size of 32KB, and a large size of 128KB.

Before entering the discussion of the results, we recall that ARC and its competitors are essentially synchronization algorithms, hence we cannot expect to achieve linear (or close to linear) scalability of throughput values depending on the variation of the number of threads. Rather, the main target objective is the one of reducing the negative effects of synchronization on throughput (and on its fall down) while moving to higher thread counts.

By the plots we see how both ARC and RF outperform the other solutions at any thread count. For small register size ARC outperforms RF as soon as the thread count is increased beyond the value 4. Also, ARC outperforms RF at any thread count for other register sizes, providing up to an order of magnitude better throughput. The reason for this behavior is that RF executes an RMW instruction (`FetchAndOr`) upon any read, while ARC executes RMW instructions 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.

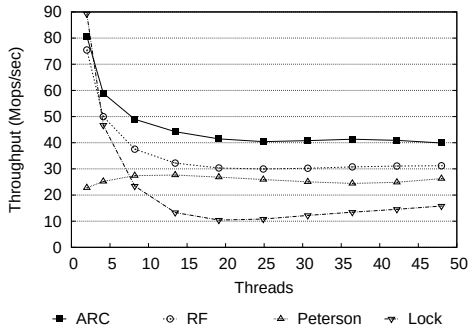
In Figure 2 we report the throughput values that have been observed when running on top of the virtualized platform with 40 vCPUs. These data confirm what we already saw for executions on the physical machine, with the additional indication that ARC performs better than RF even with minimal thread counts and for all the register sizes, which indicates how the avoidance of the execution of RMW instructions upon read operations in scenarios where newer writes were not serialized before the reads allows to favor performance even more than what happens with deploys on the 48 CPU-core physical machine. Moreover, with respect to the execution on the physical machine, all the wait-free algorithms provide a non-negligible performance speedup over the lock-based implementation. This is an indication of the benefits which can be obtained when using wait-free synchronization on virtualized architectures. Indeed, lock-based implementations can introduce an additional slow down whenever the virtualized architecture reduces the computing power allocated to the core holding the lock, due to CPU stealing by the underlying hypervisor.

In Figure 3 we report throughput data when running on the 48 CPU-core physical machine with a definitely scaled up thread count (up to 4000). In this scenario, RF could not be tested (since, as said, it supports 58 reader threads only). This test helped us to assess the performance by ARC compared to Peterson's algorithm and to the lock-based one when considering time-sharing concurrency among the threads, hence interference among them because of competition on CPU usage. By the data we see that both ARC and the lock-based algorithm are not sensible to the increase of the thread count and to the increase of the register size, even though ARC provides orders of magnitude better throughput. This is not true for Peterson's algorithm since it is based on multiple copies when performing access operations to the register. Such multiple copies are clearly adverse to performance in time-sharing concurrency deploys due to highly negative effects on locality and caching efficiency, especially for larger register size.

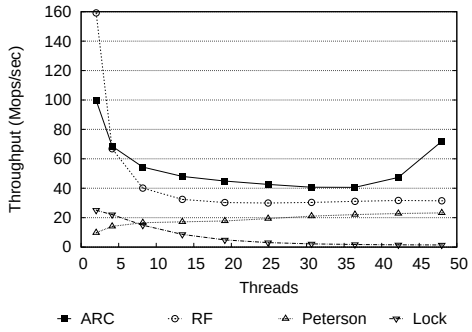
Overall, ARC delivers better performance than all the tested solutions independently of the type of deploy (physical vs virtual machine based) and of the readers' count or register size. At the same time, it still allows a huge scale-up in the number of readers compared to RF, which appeared to be the best performing literature solution (compared to Peterson's algorithm and the lock-based one) for deploys on the used physical machine.

In order to assess the performance and scalability of ARC under differentiated workload patterns, we have conducted additional experiments in which we have modified the latency of read and write operations. In particular, reads and writes have been realized in a way that mimics real processing by the threads, while trying to minimize (for all the implementations) negative effects on the cache architecture. Upon a write operation, the thread in charge of executing it creates (on stack) an array which is filled with random data. This implies that a non-minimal processing time is required to generate the new version of the data to be written to the register, which is typical of real-world applications. Upon a read operation, after having retrieved the correct snapshot of the register value, the reader scans its whole content. No actual copy of the content is made, which is likely to produce more negative effects on the cache. Anyhow, this is representative of applications which are interested in the content of the register, e.g. to produce aggregated values.

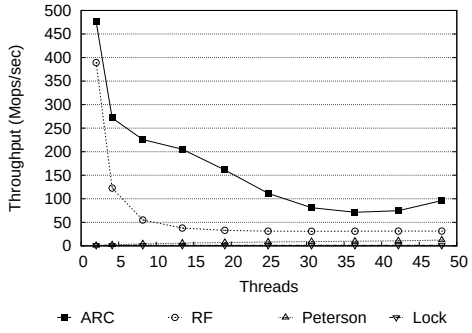
In Figure 4 we report experimental data when introducing a delay among two different write operations of around 0.5 seconds. This delay is implemented by relying on `usleep()`, which ensures that no operation is carried out by the writer thread for *at least* 0.5 seconds. This experiment is representative of scenarios where new data to be exchanged by using the register are produced less frequently (compared to the workload of the Hold-Model), yet readers continuously try to determine whether some new data have been posted—for example, like in sensors' output monitoring where a thread gets data from a sensor and



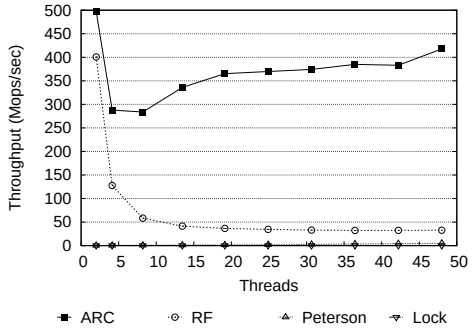
(a) 128B register size



(b) 4KB register size

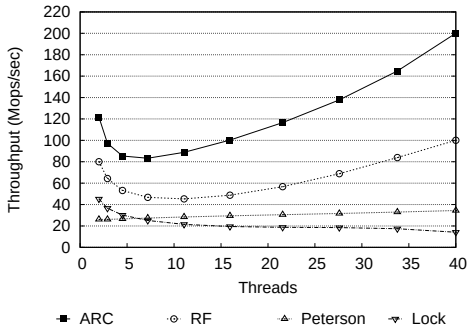


(c) 32KB register size

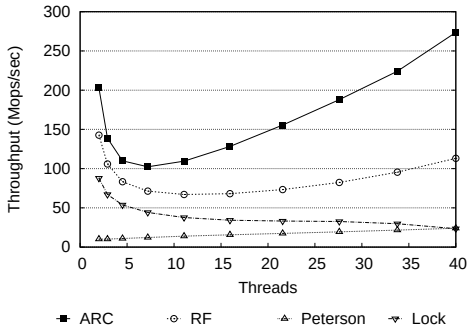


(d) 128KB register size

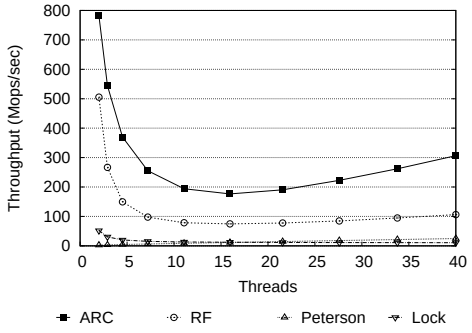
Fig. 1. Throughput with different register size values (48 CPU-core physical machine).



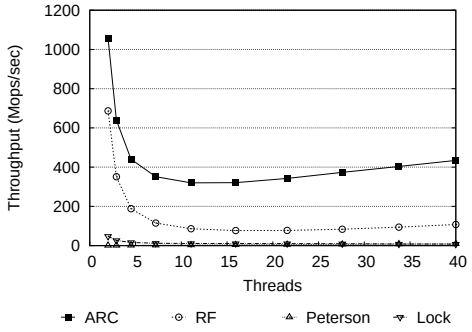
(a) 128B register size



(b) 4KB register size



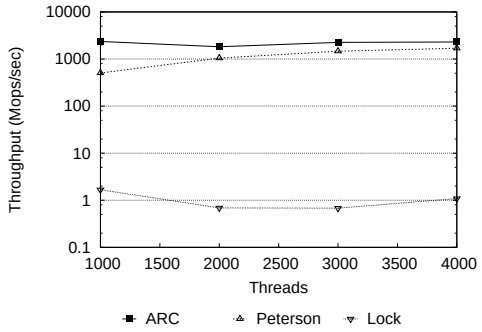
(c) 32KB register size



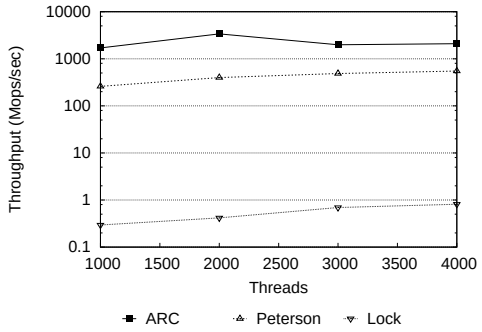
(d) 128KB register size

Fig. 2. Throughput with different register size values (40 vCPUs machine).

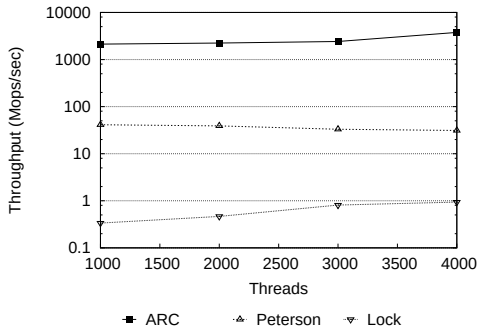
then publishes the new data value towards other threads for specific purposes, such as for allowing them to concurrently fill in input replicated data processing services [28]. In particular, the timing of the operations by the writer thread—which mimics the publishing of new data coming from some sensing device—in terms of frequency of production of new data to be posted for processing by the other threads, has been based on indications provided in the environmental monitoring project presented in [29]. By the results, we can see that ARC outperforms all the competitors. This is related to the fact that, as hinted, ARC avoids executing RMW instructions to synchronize read operations in case we are



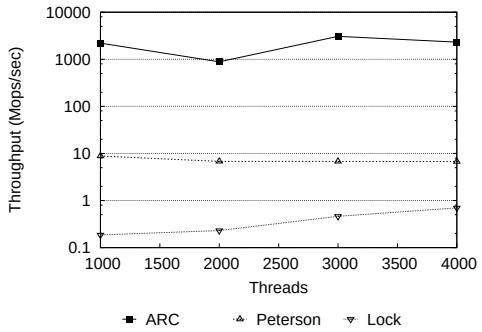
(a) 128B register size



(b) 4KB register size

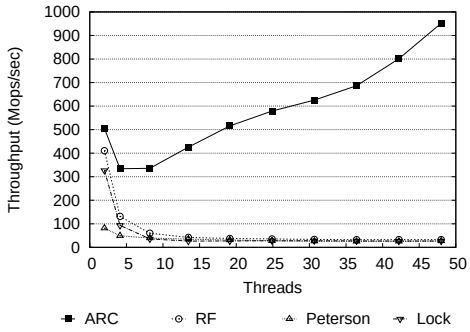


(c) 32KB register size

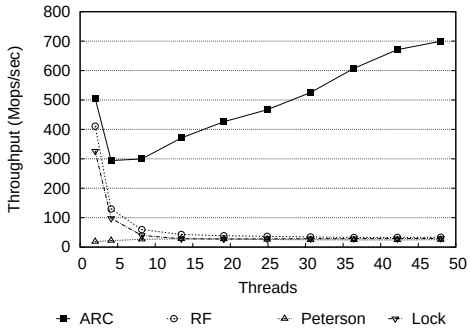


(d) 128KB register size

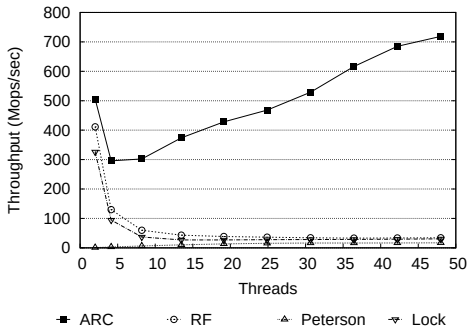
Fig. 3. Throughput with largely-increased thread counts (48 CPU-core physical machine).



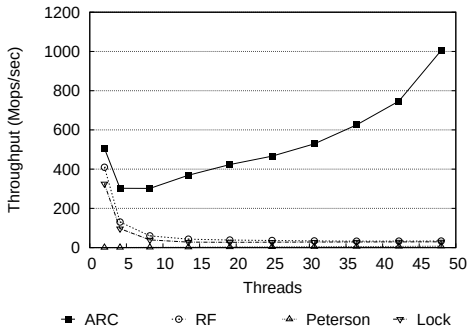
(a) 128B register size



(b) 4KB register size



(c) 32KB register size

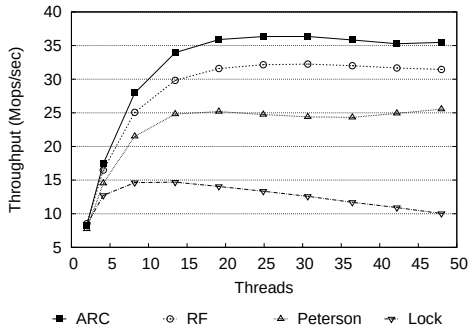


(d) 128KB register size

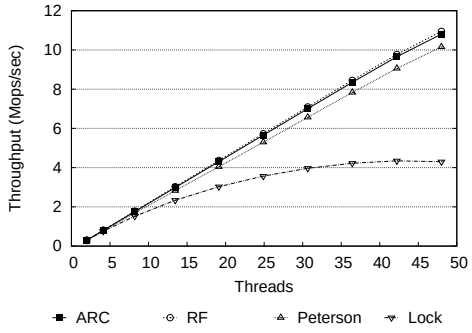
Fig. 4. Throughput with a delay of 0.5 seconds among write operations.

able to early determine that no new value has been written to the register. Since there is a non-minimal delay between two consecutive write operations, this allows us to obtain a performance speedup as high as 93% with respect to RF in the best case.

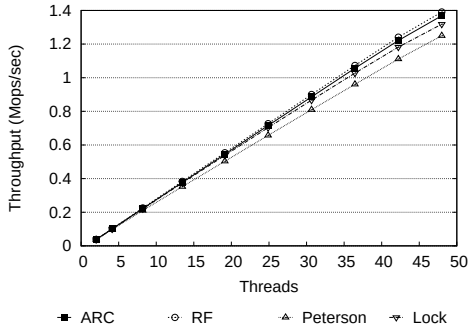
Data for a different scenario are reported in Figure 5. Here, no artificial delay is interposed among two consecutive write operations, thus sliding again towards a scenario similar to the archetypal Hold-Model. Yet, as mentioned before, both write and read operations carry out work which can be representative of real-world applications. In fact, write operations fill the written buffer with new data (although randomly generated in this test), and read opera-



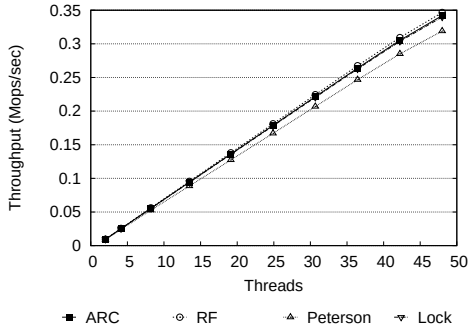
(a) 128B register size



(b) 4KB register size

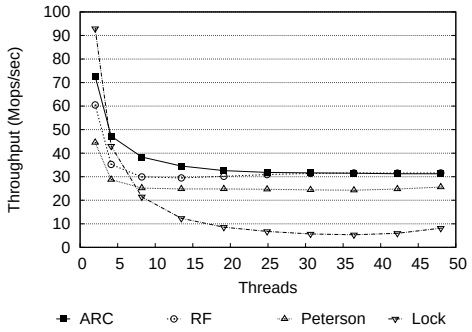


(c) 32KB register size

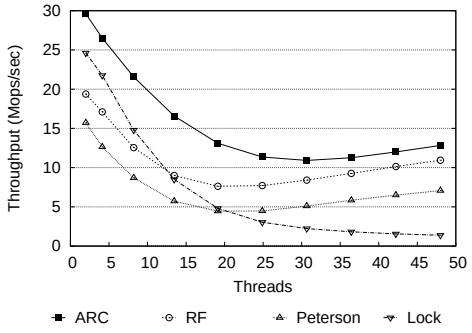


(d) 128KB register size

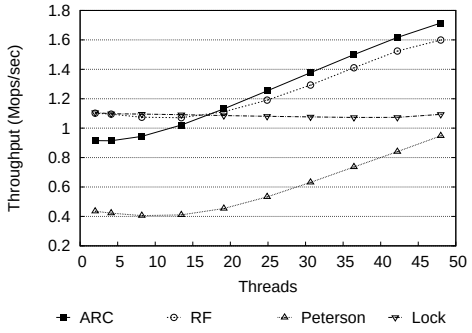
Fig. 5. Throughput when read and write operations carry out actual work, with no artificial delays.



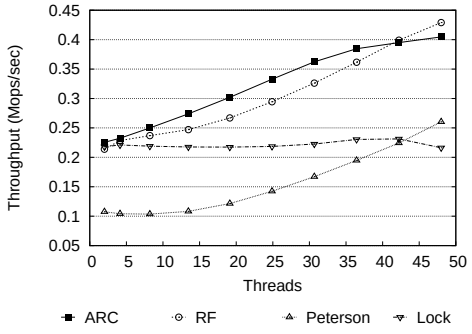
(a) 128B register size



(b) 4KB register size



(c) 32KB register size



(d) 128KB register size

Fig. 6. Throughput with high latency of reads, low latency of writes.

tions scan through the whole buffer. This scenario slows the execution of read operations wrt writes, therefore increasing the likelihood that a new value has been posted by a write operation when a new read operation is executed. Therefore, in this configuration the effect of our optimization on the read operation is reduced, forcing ARC to execute an increased number of RMW instructions.

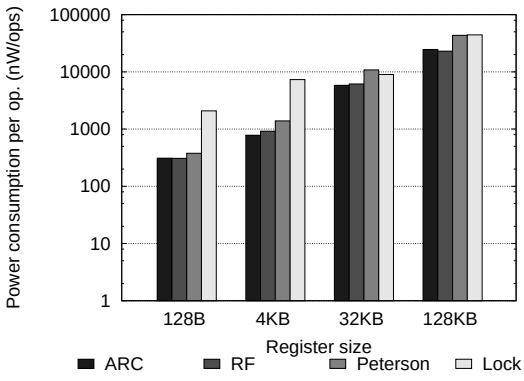
By the results, we can see that when the size of the register is increased, the number of operations per second falls down. This is an additional indication of the reduced degree of concurrency shown in this experiment, related to the fact that both read and write operations spend more time in scanning/writing the buffers. We can see that due

to this different execution pattern the performance gain by our proposal is reduced, although ARC offers anyhow a better performance with respect to all the other proposals, especially with minimal register size. This shows that our register algorithm is as well resilient to the latency of work of other nature carried out by the concurrent threads, making it a suitable solution in general real-world scenarios. We also note that for minimal register size, the achieved throughput does not scale linearly since the incidence of the relative cost of RMW instructions increases at higher thread counts.

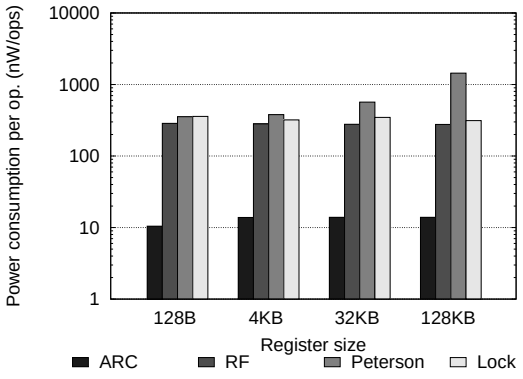
The performance of an additional scenario, which can be regarded as an adverse case for ARC, is reported in Figure 6. In this experimental setting we have reintroduced the initial "dummy" write operations—no actual content is generated, rather the same buffer is always copied as a new instance of the register value—while keeping reads which scan the whole register content. By having the writer posting the same content, our optimization of the read operation cannot take place: a new version is even more likely to be found. Additionally, the reader scans the whole content of the register, despite the fact that no new data will be actually found in it. By the results, we can see that ARC is still resilient to this negative read/write interleave pattern, showing a performance which is better than the other algorithms in most settings. The only exceptions are found with very small thread count in a few configurations of the register size, where the simple lock-based approach or RF can pay off, or for large register size and largely increased thread count, which gives rise to a scenario where the read-related optimization of ARC very unlikely materializes and RF shows slightly better performance.

To complete our experimental assessment, in Figure 7 we report data related to per-operation power consumption. These measures have been collected by relying on the "Power Gov" tool [30]. In particular, we report data collected in two opposite scenarios. Figure 7(a) is related to write operations which do not actually generate new data (the same content is always posted to the register), while read operations scan the whole content. On the other hand, Figure 7(b) is related to write operations which undergo a 0.5 seconds delay, while read operations simply retrieve a pointer to the last-written buffer. Therefore, the first configuration is related to a higher likelihood for a read to execute the whole synchronization algorithm, while in the second scenario ARC read-related optimization is likely to pay off. By the results in Figure 7(a), ARC shows power consumption that is anyhow smaller than, or close to, the one of any other algorithm. It is interesting to note that, in this scenario, Peterson's algorithm has power consumption that is even higher than the simple lock-based algorithm. As for this aspect, we recall that Peterson's algorithm relies on multiple buffers—each reader accesses at least two buffers to complete an operation. This has a non-negligible effect on the cache hierarchy (also in terms of power usage).

In Figure 7(b), we see that ARC power consumption is constant with respect to the register size, and is definitely lower than the one of the competitors. This is related to the fact that with higher likelihood readers efficiently determine that no new data has been posted to the register. Therefore, the amount of executed RMW instructions is significantly reduced, with benefits on power usage by cache controllers to access synchronization variables.



(a) Frequent Writes



(b) Delayed Writes

Fig. 7. Power consumption - runs with 48 threads.

7 Conclusions

In this article we have presented Anonymous Readers Counting (ARC), a multi-word wait-free atomic (1,N) register algorithm targeting shared-memory TSO-consistent parallel architectures. Our register 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 commonly supported by off-the-shelf architectures, by also reducing the impact of actually running RMW instructions compared to the reference literature proposal in [3], which also has the disadvantage of handling up to 58 readers only. We have also shown how to adapt ARC to manage the multiple writers case—the so called (M,N) register—still in wait-free manner and with essentially no change along the critical path of the operations by the concurrent threads that write/read the register. The performance benefits from our proposal compared to literature approaches have been shown via a study based on deploys of the compared register implementations on both a parallel physical machine and a virtualized one. We have also provided a proof of correctness of our register algorithm.

REFERENCES

[1] M. Li, J. Tromp, and P. M. B. Vitányi, "How to share concurrent wait-free variables," *Journal of the ACM*, vol. 43, no. 4, pp. 723–746, 1996.

[2] M. P. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 124–149, 1991.

[3] 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.

[4] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, "Intel QuickPath interconnect architectural features supporting scalable system architectures," in *Proceedings of the18th IEEE Symposium on High Performance Interconnects*, pp. 1–6, 2010.

[5] R. J. Safranek and M. J. Moravan, "QuickPath interconnect: Rules of the revolution," *Dr. Dobb's Journal*, 2009.

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

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

[8] A. Silberschatz and P. Galvin, *Operating System Concepts*. Addison-Wesley Publishing Company, 1994.

[9] G. Barrett, "Model checking in practice - The T9000 virtual channel processor," in *Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pp. 129–147, 1993.

[10] 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.

[11] G. L. Peterson, "Concurrent Reading While Writing," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 1, pp. 46–55, 1983.

[12] H. Simpson, "Four-slot fully asynchronous communication mechanism," *IEE Proceedings E (Computers and Digital Techniques)*, vol. 137, no. 1, pp. 17–30, 1990.

[13] S. Haldar and K. Vidyasankar, "Constructing 1-writer multireader multivalued atomic variables from regular variables," *Journal of the ACM*, vol. 42, no. 1, pp. 186–203, 995.

[14] P. M. B. Vitányi, B. Awerbuch, P. Vitanyi, and B. Awerbuch, "Atomic shared register access by asynchronous hardware," in *27th Annual Symposium on Foundations of Computer Science*, pp. 233–243, 1986.

[15] J. H. Anderson and M. Moir, "Universal constructions for multi-object operations," in *Proceedings of the 14th annual ACM Symposium on Principles of Distributed Computing*, pp. 184–193, 1995.

[16] P. Fatourou and N. D. Kallimanis, "A highly-efficient wait-free universal construction," in *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, p. 325, 2011.

[17] Z. Aghazadeh, W. Golab, and P. Woelfel, "Making objects writable," in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pp. 385–395, 2014.

[18] L. Zhu and F. Ellen, "Atomic snapshots from small registers," in *Proceedings of the 19th International Conference on Principles of Distributed Systems*, 2015.

[19] M. P. Herlihy, "Impossibility and universality results for wait-free synchronization," in *Proceedings of the 7th annual ACM Symposium on Principles of Distributed Computing*, pp. 276–290, 1988.

[20] LINUX.ORG, "`https://www.linux.org/`."

[21] A. Matveev, N. Shavit, P. Felber, and P. Marlier, "Read-log-update: a lightweight synchronization mechanism for concurrent programming," in *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 168–183, 2015.

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

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

[24] 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.

[25] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, 1979.

[26] J. Burns and N. A. Lynch, "Mutual exclusion using invisible reads and writes," in *Proceedings of the 18th Annual Allerton Conference on Commun., Control, and Computing*, pp. 833–842, 1980.

[27] J. G. Vaucher and P. Duval, "A comparison of simulation event list algorithms," *Comm. of the ACM*, vol. 18, no. 4, pp. 223–230, 1975.

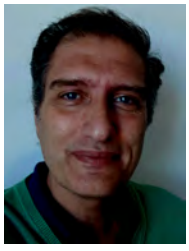
[28] P. Romano, D. Rughetti, B. Ciciani, and F. Quaglia, "APART: Low cost active replication for multi-tier data acquisition systems," in *Proceedings of the 7th EEE International Symposium on Network Computing and Applications*, pp. 1–8, 2008.

[29] B. Ciciani, P. Di Sanzo, U. Nanni, F. Quaglia, and F. Sarracco, "Osservambiente - a Project for Territorial Governance," in *Proceedings of the VI Conference of the Italian Chapter of AIS*, 2009.

[30] https://software.intel.com/sites/default/files/managed/13/3d/power_gov.rev72.tgz.

Mauro Ianni is a PhD student at Sapienza, University of Rome, and is a member of the group High Performance and Dependable Computing Systems research group at the same institution. He achieved the Bachelor's degree in Computer Engineering in 2012 and the Master's degree in Distributed Systems and Computer Architectures in 2015. His research activities focus on the development of methodologies and techniques to ensure the correctness of operations in concurrent environments avoiding explicit synchronization, and applications to support data processing on massively parallel environments.

Alessandro Pellegrini has received the PhD in Computer Engineering at Sapienza, University of Rome in 2014. His main research topic is simulation on parallel and distributed architectures, a field where he has published more than 50 among books, book chapters, journal articles, and international conference proceedings papers. In 2015 he has won the Sapienza prize for the best PhD thesis of the year. He has worked as a researcher at some national and international research centers, such as CINI, CINFAI and IRIANC. His additional competence spans from compilers to high-performance systems. He has actively contributed to the development of open-source applications which are currently used at some research centers in Europe. He has served as TPC member or organizing member of several International Conferences.



Francesco Quaglia received his MS in Electronic Engineering in 1995 and his PhD in Computer Engineering in 1999, both from Sapienza University of Rome, where he has worked as Assistant Professor and then Associate Professor from September 2000 till June 2017. Currently he works as a Full Professor at the University of Rome Tor Vergata. His research interests include parallel and distributed computing systems and applications, operating systems, high performance computing and fault tolerance. In these areas, he has authored (or coauthored) more than 180 technical articles. He has been program or general chair for prestigious international conferences, and has been (unit) coordinator for national and EU projects addressing topics in the above areas. He has been awarded the best paper five times at re-known international conferences. He won the Future Grid 2012 Project Challenge Award thanks to technical results in the area of distributed/replicated transactional systems achieved within the Cloud-TM FP7 Project. Currently he is an Associate Editor of ACM Transactions on Modeling and Computer Simulation.