SABRes: Atomic Object Reads for In-Memory Rack-Scale Computing

Alexandros Daglis, Dmitrii Ustiugov, Stanko Novaković
Edouard Bugnion, Babak Falsafi
EcoCloud, EPFL
Email: firstname.lastname@epfl.ch

Boris Grot
University of Edinburgh
Email: boris.grot@ed.ac.uk

***Abstract*—Modern in-memory services rely on large distributed object stores to achieve the high scalability essential to service thousands of requests concurrently. The independent and unpredictable nature of incoming requests results in random accesses to the object store, triggering frequent remote memory accesses. State-of-the-art distributed memory frameworks leverage the one-sided operations offered by RDMA technology to mitigate the traditionally high cost of remote memory access. Unfortunately, the limited semantics of RDMA one-sided operations bound remote memory access atomicity to a single cache block; therefore, atomic remote object access relies on software mechanisms. Emerging highly integrated rack-scale systems that reduce the latency of one-sided operations to a small multiple of DRAM latency expose the overhead of these software mechanisms as a major latency contributor.**

This technology-triggered paradigm shift calls for new one-sided operations with stronger semantics. We take a step in that direction by proposing SABRes, a new one-sided operation that provides atomic remote object reads in hardware. We then present LightSABRes, a lightweight hardware accelerator for SABRes that removes all atomicity-associated software overheads. Compared to a state-of-the-art software atomicity mechanism, LightSABRes improve the throughput of a microbenchmark atomically accessing 128B-8KB objects from remote memory by 15-97%, and the throughput of a modern in-memory distributed object store by 30-60%.

1. Introduction

Large-scale online services operate on massive data with tight latency constraints. To meet these requirements, data is kept in memory-resident data stores (such as key-value stores) distributed across hundreds of servers. With each incoming user request touching several data objects spread across multiple servers, frequent and fine-grain inter-server communication becomes unavoidable. Unfortunately, remote memory access over conventional networking is orders of magnitude slower ($\sim$1000$\times$) than local memory access, significantly diminishing the benefits of keeping the data in memory. Several modern software frameworks for in-memory distributed computing show dramatic performance improvements by moving from conventional to RDMA networking [12], [21], [31], [32], [45]. RDMA technology enables fast direct access to remote memory by offering one-sided operations to completely bypass the remote CPU.

Existing RDMA technologies such as InfiniBand can deliver remote memory access at a latency as low as 10-20$\times$ of local memory access (1-2μs vs. 60-100ns). Emerging rack-scale systems further shrink this gap through tight integration, lean user-level protocols, and high-performance fabrics, bringing remote memory access latency just within a small factor over local memory access [19], [29], [34].

One-sided operations have semantic limitations, which is a direct consequence of their DMA-based implementation on the remote end: while each individual cache block can be accessed atomically, there are no guarantees for larger accesses straddling multiple cache blocks. To overcome this limitation, data management systems leveraging one-sided operations employ software mechanisms such as locks or optimistic concurrency control to enforce atomic remote object accesses [12], [32], [45].

Providing object atomicity in software will gradually become a performance limiter as modern fabrics improve the communication latency and bandwidth. Indeed, our study shows that the state-of-the-art software mechanism delivering atomic object accesses in FaRM [12] accounts for up to 50% of the end-to-end remote memory access latency for large objects (8KB) on Scale-Out NUMA [34]. Consequently, providing atomic remote object access becomes a first-order performance concern calling for architectural support to replace the costly software mechanisms.

Since remote object reads represent the most frequent remote memory operation, introducing a one-sided hardware primitive with the semantics of an *atomic remote object read* is critical to the performance of future tightly integrated rack-scale systems. To cover this requirement, we introduce a new one-sided operation type, called *SABRe* (Single-site Atomic Bulk Read) and investigate the design space to provide this new operation in hardware. We then describe an implementation of SABRes and illustrate major performance improvements as compared to a state-of-the-art software mechanism for atomic remote object reads.

Our contributions include:

- A study of the overhead of providing remote object read atomicity in software on emerging rack-scale systems. We show that this overhead accounts for a measurable

978-1-5090-3508-3/16/$31.00

fraction of the end-to-end remote memory access latency: 10-50% for 128B-8KB objects.
- A design space exploration for hardware SABRes, a new type of one-sided operation with stronger semantics than any existing one-sided primitive.
- LightSABRes, a lightweight and high-performance implementation of SABRes, integrated into the chip's coherence domain to detect object atomicity violations. LightSABRes completely remove the software overhead associated with remote object read atomicity.
- An evaluation of LightSABRes on a state-of-the-art rack-scale system, Scale-Out NUMA [34]. We show significant throughput improvements for atomic remote reads of 128B-8KB objects: 15-97% with a microbenchmark and 30-60% with a key-value store on a full software stack for in-memory distributed computing.

The rest of the paper is organized as follows. We present some essential background in §2, explore the design space for providing SABRes in §3, and describe our implementation of LightSABRes in §4. We then couple LightSABRes with Scale-Out NUMA as a case study in §5, detail our methodology in §6, and evaluate the resulting system in §7. Finally, we discuss related work in §8 and conclude in §9.

2. Background

Modern frameworks for in-memory distributed computing leverage RDMA's low communication latency to dramatically improve the performance of remote memory access [12], [21], [32], [45]. These frameworks build in-memory object stores that take advantage of one-sided operations to deliver fast access to remote objects. Such object stores are the backbone of large-scale online services.

For applications that operate on structured data, the granularity of an operation (and also the minimum unit of transfer when accessing remote memory) is the object. The size of these objects is application-specific, and can range from a few bytes to several kilobytes [26]. Unfortunately, RDMA technology relies on PCIe DMA to transfer data between the memory and the network, and therefore its remote memory access semantics are limited to read, write, and cache-block-sized atomic operations, such as remote CAS. The latter only provide atomic access to a memory region not exceeding a single cache block in size. No existing hardware mechanism can provide atomic access to larger memory regions; thus, the challenge of accessing objects atomically falls on the software.

2.1. Atomic one-sided operations

Several modern frameworks for in-memory distributed computing rely on one-sided RDMA operations (e.g., Pilaf [32], FaRM [12], DrTM [45]). One-sided operations deliver fast access to remote memory by completely avoiding remote CPU involvement, but offer limited semantics. In most cases, one-sided operations are only used for reads, while writes are sent to the data owner over an RPC. This common design choice simplifies software design and is motivated by the read-dominated nature of most applications.

To the best of our knowledge, the only system using one-sided operations for both reads and writes is DrTM [45]. DrTM uses HTM as an enabler for one-sided writes, relying on it to detect local conflicts with incoming remote writes and abort conflicting local reads. While DrTM introduces an interesting design point, we focus on the common case of *one-sided read operations*. Because HTM functionality is bounded to its local node's coherence domain, it cannot be directly used for atomic multi-cache-block remote reads.

Modern frameworks rely on software techniques to complement the limited semantics of one-sided operations, which only offer cache-block-sized atomicity. A defining characteristic for these techniques is the employed concurrency control (CC) method: locking vs. optimistic concurrency control (OCC). Combining locking with one-sided reads is simple. Each object in the data store has an associated lock. When a node requires atomic access to a remote object, it issues a first one-sided (cache-block atomic) RDMA CAS operation to acquire the remote object's lock, followed by another one-sided operation to access the object atomically—locking prevents any conflicts.

However, remote lock acquisition comes with two drawbacks. First, it increases the latency of remote memory access by an additional network roundtrip. Second, it introduces fault-tolerance concerns, as a node's failure may result in deployment-wide deadlocks, turning the RDMA cluster into a single failure domain and thus jeopardizing the traditional high resilience of scale-out deployments. The latter concern can be addressed for reads by replacing conventional locks with lease locks, as illustrated by DrTM. Unfortunately, lease locks are sensitive to clock skew across the deployment's machines, and their duration can significantly impact concurrency and abort rates.

OCC addresses the shortcomings of remote locking. Driven by the observation that most workloads are read-dominated, and hence the probability of a conflict is low, OCC relies on conflict detection rather than conflict prevention for high performance (Pilaf [32], FaRM [12]). Since hardware only provides cache-block-sized atomicity, remote reads are paired with ad hoc software-based mechanisms for conflict detection, which do not come for free. For instance, Pilaf [32] embeds a checksum in each object's header as additional metadata. The checksum is recomputed after every update, and remote readers compute the checksum of the object's data to compare it to the object's checksum—a mismatch indicates an atomicity violation. Unfortunately, while conceptually simple, the checksum mechanism is expensive, as the cost of CRC64 is about a dozen CPU cycles per checksummed byte [32]. For KB-sized objects, this overhead can grow to tens of thousands of CPU cycles (i.e., several microseconds) per object transfer.

FaRM [12] introduces the more efficient approach of per-cache-line versions: every object has a 64-bit version in its header, whose l least significant bits are replicated in a per-cache-line header. Writers update all versions upon an object update, and readers compare all cache-line versions

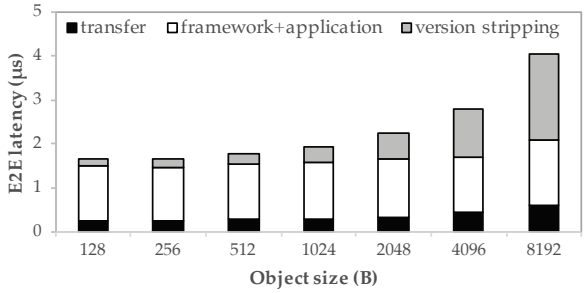


Figure 1: End-to-end remote object read latency using the per-cache-line-version software atomicity check mechanism on FaRM over soNUMA.

to detect atomicity violations before consuming the data. While computationally cheaper than checksums, per-cache-line versions still introduce measurable CPU overhead for both readers and writers. More importantly, per-cache-line versions prevent zero-copy object transfers: before the application can use the object, the CPU has to extract the clean data into a buffer by stripping off the embedded per-cache-line versions. This overhead applies to all types of read/write accesses, both local and remote. Despite the overhead, FaRM's per-cache-line versions mechanism is the state-of-the-art approach to provide *optimistic* and *atomic* one-sided reads from remote memory.

2.2. Emerging rack-scale systems

While RDMA is the leading product in providing fast inter-node communication and remote memory access, its performance is ultimately capped by the latency overhead of the PCIe interface [34]. With single-cache-line RDMA reads exceeding 1μs in latency, the latency of accessing remote memory alone dwarfs the latency of consequent local memory operations, such as the post-transfer data extraction and version checks required when using FaRM's per-cache-line versions technique, which may only account for a few hundred nanoseconds. Thus, FaRM's design choice does not effectively impact the end-to-end latency of one-sided RDMA reads.

However, RDMA technology is evolving, moving away from PCIe and towards tightly integrated solutions. For instance, AppliedMicro's X-Gene 2 [28] and Oracle's Sonoma [29] integrate an RDMA controller on chip. The trend towards tight integration is not limited to the chip level. In fact, recent technological advancements have led to the emergence of tightly integrated chassis- and rack-scale systems, such as HP's Moonshot [18] and The Machine [19], Oracle Exadata [35], and AMD SeaMicro [11]. These systems interconnect a large number of servers, each with an on-chip network interface (NI), using a supercomputer-like lossless fabric. NI integration and short intra-rack communication distances help reduce communication delays. At the same time, research proposals (e.g., Firebox [2], Scale-Out NUMA [34]) show how sub-μs remote memory access is achievable through the combination of lean network protocols, tight integration, and contained physical scale. We envision that emerging rack-scale systems will soon adopt such lightweight network stacks, which, combined with tightly integrated SoCs, will significantly improve the performance of remote memory access in terms of both latency and bandwidth as compared to existing RDMA solutions. In such an environment, any software overhead added to the bare remote memory access latency imposed by the underlying hardware will perceivably increase the end-to-end latency.

2.3. The case for SABRes

In the context of emerging tightly integrated rack-scale systems, we evaluate the performance impact of software-based atomicity mechanisms. As a case study, we use Scale-Out NUMA (soNUMA) [34] and run a key-value store on top of FaRM [12]. We simulate two directly connected soNUMA nodes to measure the latency breakdown of one-sided remote reads. Object atomicity is achieved through FaRM's per-cache-line versions mechanism. Simulation parameters can be found in §6.

Fig. 1 shows the end-to-end latency breakdown of an atomic remote object read. For every object size, we break down the latency into three components: the soNUMA transfer time, the time spent in the FaRM framework and application code, and the time spent by the core extracting useful data from the transferred object, by stripping off and comparing the per-cache-line versions to check for atomicity violation. We observe that the latency of one-sided reads over soNUMA starts at just 3-4x of local memory access and scales sublinearly with object size, due to soNUMA's high-bandwidth fabric. In contrast, while the software atomicity check latency is negligible for small objects ($\sim$10% for 128B objects), it scales almost linearly with object size and thus quickly outgrows the soNUMA transfer latency, accounting for 50% of the end-to-end latency for 8KB objects. Furthermore, a fraction of the latency goes to FaRM buffer management, which is necessary for storing the transferred data, before it is cleaned up and moved to the application's buffer.

In this work, we introduce a new *Single-site Atomic Bulk Read (SABRe)* one-sided primitive in hardware that removes the atomicity-associated software overhead and enables zero-copy transfers, by obviating the need for intermediate buffering.

3. SABRe Design Space

In this section, we investigate the design space for providing the essential hardware support for SABRes. We consider systems that feature on-chip integrated protocol controllers supporting one-sided remote memory operations.

3.1. Destination-side concurrency control

Table 1 summarizes the design space for atomic remote object access, with or without hardware support. In our

	Source	Destination
Locking	DrTM [45]	SABRes
OCC	FaRM [12], Pilaf [32]	

TABLE 1: Design space for one-sided atomic object reads.

taxonomy, the terms *source* and *destination* refer to request processing location rather than data location. Under that definition, all software-based approaches leveraging one-sided operations essentially implement *source-side* CC since the destination side's CPU is not involved. DrTM relies on acquiring remote locks, with locking explicitly controlled at the source prior to accessing the remote object's data. FaRM and Pilaf implement different OCC mechanisms to enforce atomicity, but as the source has to perform post-transfer atomicity checks, both are source-side mechanisms.

Introducing hardware support expands the design space, with possible source-side or destination-side accelerators. For example, one can easily envision source-side hardware accelerators that deal with hardware checksums or per-cache-line versions. However, such an approach has a number of weaknesses. First, cache-block-sized replies with payloads can arrive out of order. Depending on the mechanism, these replies might need to first be reordered, requiring intermediate buffering (e.g., in the case of checksums). Second, the application's whole data store needs restructuring just to embed the necessary per-object metadata that enable atomicity checks for one-sided remote operations. Such restructuring also affects the performance of all local operations (reads & writes), as they have to comply with the modified data layout's rules: readers might need to unpack data before consumption, writers need to always update corresponding metadata as well. Ultimately, the weakness of source-side mechanisms is that they are limited to post-transfer atomicity checks and thus require additional metadata embedded in—and always transferred with—the requested remote object.

In contrast, destination-side hardware support offers more appealing opportunities. Providing CC directly at the destination is a natural option; this is where the target data is located and, thus, where synchronization between concurrent accesses to that data occurs. Therefore, destination-side CC offers higher flexibility and efficiency, such as leveraging local coherence for online atomicity violation detection and obviating the need to maintain and transfer any additional metadata for post-transfer validation at the source. For instance, locking directly at the destination cancels both drawbacks of remote locking (i.e., increased latency and fault-tolerance concerns). Similarly, reading data optimistically while actively monitoring atomicity at the destination obviates the need for restructuring the data store to embed OCC-specific metadata, and also allows for early conflict detection.

Overall, destination-side CC comes with many desirable properties, which trump source-side alternatives. Therefore, our hardware extensions for SABRes target Table 1's rightmost column, representing the first destination-side CC solution solely based on one-sided operations.

3.2. Design goals

Given the advantages of destination-side CC, we now define the three design goals (DG) necessary for an efficient SABRe hardware design:

- [DG1] *Minimal single-SABRe latency.*
- [DG2] *High inter-SABRe concurrency*. The mechanism should be able to utilize all the available bandwidth even with a multitude of small SABRes.
- [DG3] *Low hardware complexity/cost* (e.g., no modifications to the chip's coherence protocol).

A straightforward and efficient approach to implement SABRes is lock acquisition at the destination. Since objects typically have a header with a lock for synchronization between local threads, the controller can acquire the lock as any other local thread. To support high reader concurrency, shared reader locks are essential, yet only add minimal complexity to the locking logic.

For read-dominated applications, OCC is typically preferable to locking. For that reason, many modern software frameworks, such as key-value stores and in-memory DBMSs, do not employ reader locks, but rely on optimistic reads for high reader throughput (e.g., [12], [25], [30], [43], [44]). To enable optimistic reads, objects have a version in their header, which is incremented at the beginning and at the end of each update. To determine a read's atomicity, the controller simply compares the version's value before and after the read. Enhancing the protocol controller at the destination for optimistic reads is also quite simple: instead of acquiring an object's lock, the controller can at any time assess the object's state by reading the object's version.

The biggest drawback of a naive implementation of either mechanism for hardware SABRes (locking, or OCC using version checks) is the requirement for a serialized first access to read the version or acquire the lock prior to any data access. In the general case when the target object is in memory, this requirement can significantly increase the object read latency by exposing the full latency of that first memory access (i.e., $\sim$60-100ns), incurring a considerable latency overhead especially for small objects. To illustrate, on a tightly integrated system such as soNUMA, this serialization can increase the end-to-end latency of a two-cache-block SABRe by up to 40% (details in §7.1). In the case of version comparison, an additional serialized load to re-read the object's version after all data has been read is also required. However, the latency overhead of this second load is less critical, as it will likely hit in an on-chip cache.

Violating the *read-version-then-data* (or *acquire-lock-then-read-data*) serialization to avoid exposing that latency penalty can result in undetected atomicity violations. Fig. 2 illustrates a potential race condition that may arise if we overlap the version read with data read. In this example, we assume that the protocol controller receives a remote read request for an object that spans two cache blocks and

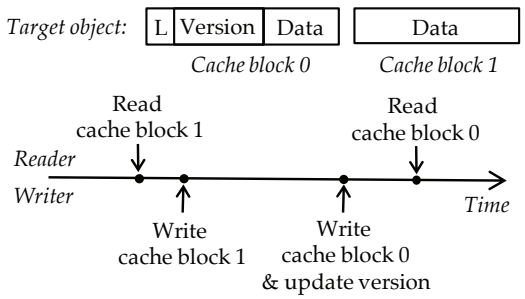


Figure 2: Reader-Writer race example.

that it implements OCC using the object's header version (the example equally holds in the case of locking). Cache block 0 contains the object's header, with the corresponding lock and version, and a writer currently holds the object's lock. If the controller issues a read for cache blocks 0 and 1 concurrently, the read for cache block 1 may complete first, as any reordering can occur in the memory subsystem and on-chip network. Then, the writer modifies cache block 1, updates the object's version and frees the lock (cache block 0). After this intervention, the controller's read for cache block 0 also completes, finding a free lock. At this point, the controller has no means of detecting the writer's intervention and wrongly assumes that the object has been read atomically, while in practice it has retrieved the latest value of cache block 0 and an old value of cache block 1. Reading the object's version *before* issuing any other read operation, though, guarantees that such races causing transparent atomicity violations cannot occur.

A careful implementation of the simple hardware SABRe mechanisms mentioned above can satisfy DG2 and DG3 (i.e., high inter-SABRe concurrency and low hardware complexity), but not DG1 (i.e., minimal single-SABRe latency), because of the serialization limitation. We can break the *read-version-then-data* problem by leveraging speculation techniques. The tight integration of the protocol controllers with the chip also implies integration into the chip's coherence domain. This integration enables a variety of options regarding atomicity enforcement mechanisms. Speculation techniques proposed for relaxing memory ordering (e.g., fence speculation) [3], [16], [33], or conflict detection and resolution mechanisms employed by HTM could be directly applicable to register and guard a SABRe's address range during its lifetime. However, those mechanisms are unnecessarily complex and contradict DG3.

Our key insight is that SABRes require considerably simpler functionality than HTM or other sophisticated speculative structures employed by aggressive cores to relax memory order. First, SABRes only involve reads and no writes. Second, SABRes naturally come with software-provided characteristics that can simplify hardware requirements; that is, SABRes are by definition accesses to structured data that comprise objects in a data store rather than accesses to arbitrary memory locations. Every object typically features a header with associated metadata, such as a lock and/or a version, and a range of sequential addresses containing data. Writers update this header accordingly upon each write to the object. We can thus expose these semantics to the hardware, and rely on a hardware-software contract to simplify the hardware.

3.3. Safely overlapping lock & data access

We now leverage our insights from §3.2 to design a lightweight hardware mechanism that safely overlaps an object's lock/version access and data read, meeting DG1. It is possible to hide the serialization latency and read all data in parallel instead, thus extracting maximum memory-level parallelism (MLP), as long as we provide a mechanism to detect any data atomicity violation that may occur before the completion of the object's first version read or lock acquisition. Since memory accesses can be reordered by the memory subsystem, requested data may return in any order. We define the time between issuing an access to the SABRe's first cache block, which contains the object's version or lock, and its completion, as that SABRe's *window of vulnerability*. Within that window, all data are speculatively read, as it is unknown whether the read operation is racing against a concurrent write to the same object, risking a transparent atomicity violation as in Fig. 2's example.

We rely on the integration of the protocol controller in the chip's coherence domain to detect atomicity violations during this window of vulnerability. Given that a SABRe comprises a sequence of reads to consecutive addresses, the mechanism only needs to snoop coherence traffic for an *address range* rather than a set of independent addresses. At the high level, such range tracking can be trivially implemented by a structure that just keeps track of a SABRe's starting address and length, allowing for simple indexed lookups through simple base-and-offset arithmetic. Using this structure, the loads comprising a SABRe can be performed in parallel, exploiting maximum MLP. The critical addresses are trivially captured as an address range and are snooped upon each reception of a coherence invalidation message during the window of vulnerability. An invalidation matching an address of an already read block triggers an abort of the corresponding SABRe.

Implementing such address range snooping structure in hardware is much simpler than an out-of-order processor's load-store queue, or an address resolution buffer [14], [15], [39]: no dynamic memory disambiguation or associative searches, within or across different address range snooping structures are required. We provide an implementation of the proposed mechanism in the following section.

4. LightSABRes

In this section we describe *LightSABRes*, an implementation of a destination-side CC mechanism for SABRes that performs address range snooping using stream buffers. We assume a generic on-chip protocol controller for one-sided operations integrated into the chip's coherence domain.

4.1. Address range snooping implementation

We implement address range snooping by leveraging an adaptation of stream buffers [20], illustrated in Fig. 3. Every inbound SABRe request is associated with a stream buffer; starting from the SABRe's *base physical address*, each SABRe cache block is mapped to an entry of the associated stream buffer. Since all blocks comprising a SABRe are consecutive, issued loads for the same SABRe map to consecutive stream buffer slots (with the exception of SABRes spanning two non-consecutive physical pages). The stream buffer holds the range of addresses touched by the controller during the window of vulnerability.

Integration of such stream buffers with the protocol controller allows overlapping the object's lock/version access and data read, thus enabling maximum MLP for a single SABRe even during the window of vulnerability. The controller can keep pushing consecutive cache-block-sized read requests to the memory hierarchy as long as (i) the SABRe's associated stream buffer is deep enough to contain all the outstanding loads; and (ii) there is no boundary crossing between two non-consecutive physical pages. If the controller hits any of these two limitations while issuing loads for a SABRe, that SABRe simply needs to stall, without any correctness implications. Once the window of vulnerability is over (i.e., the version/lock is accessed), the stream buffer is not useful anymore and reading the object's data can seamlessly continue without the previous two limitations. Page boundary crossing during the window of vulnerability is an infrequent event that does not raise performance concerns, especially given the common RDMA/soNUMA practice of using superpages for the memory regions exposed to the global address space (e.g., [12]).

A stream buffer's entries represent a sequence of *loads to consecutive physical memory addresses*. With the exception of the head entry, stream buffer entries do not store an address. Instead, each entry's corresponding address is deduced as a simple addition of the stream buffer's associated base address and its location offset. This property provides a cheap lookup mechanism through simple indexing rather than associative search. Data replies arriving from the memory hierarchy are not stored in the stream buffer either, but are directly sent back to the requester by the protocol controller.

In Fig. 3's example, white stream buffer entries are currently unused, gray entries denote cache-block accesses that have been issued to the memory hierarchy and await a reply, while black entries have already received a reply and the payload has already been sent back to the requester. The controller issues the third cache-block read request for *SABRe 0*. At the same time, a coherence invalidation message is received for *SABRe 1*'s fifth cache block. Since *SABRe 1*'s head cache block has not yet been accessed, this invalidation indicates a possible race condition with a writer, so *SABRe 1* will abort. In contrast, *SABRe n* does not abort upon reception of an invalidation for its last stream buffer

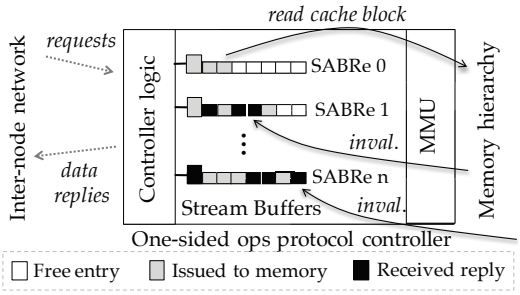


Figure 3: Stream buffer to safely overlap lock & data access.

entry, as it has already accessed the head entry's block; this invalidation must have been triggered by an eviction.

The key insight regarding stream buffer provisioning that makes our mechanism lightweight and scalable is that both the number and depth of required stream buffers is orthogonal to the SABRe's length. Sizing is only a function of the memory hierarchy and the target peak bandwidth of the controller that is enhanced with LightSABRes. The number of stream buffers defines the maximum number of concurrent SABRes the controller can handle; there should be enough stream buffers to allow the controller to utilize its full aggregate bandwidth even for the smallest SABRes (i.e., two-cache-block SABRes). The depth of the stream buffers affects the latency of each SABRe. The controller can keep pushing cache block load requests for a SABRe, as long as there are available slots in that SABRe's corresponding stream buffer *or* the target object's version has been read (or, in the case of locking, the object's lock has been acquired). Thus, the stream buffer's depth should be sufficient to allow pushing data load requests to the memory subsystem at the controller's maximum bandwidth, until the first request to access the object's version/lock completes.

4.2. System integration

As discussed in §3.2, several modern software frameworks rely on OCC, allowing readers to optimistically proceed without acquiring any locks, as conflicts are expected to be rare and retries are cheap. Without loss of generality, we will focus the implementation description of LightSABRes on an OCC mechanism. The same principles are applicable to locking; in fact, the same implementation with minimal modifications can be used for both locking and OCC.

We assume that the software maintains versions for CC similar in philosophy to Masstree's [30] object versions. Each object has a version in its header. Writers increment the version to acquire exclusive access to an object, and increment it again once they are done with their changes. Thus, an odd version indicates a locked object, and an even version indicates a free object. This is functionally equivalent to having a lock acquired before updating an object, and a version incremented before the lock is freed again. Therefore, without loss of generality, we assume that writers use the odd/even version mechanism for updates.

Fig. 4 shows a LightSABRes-enhanced protocol controller pipeline, which we refer to as *R2P2*. The key entity

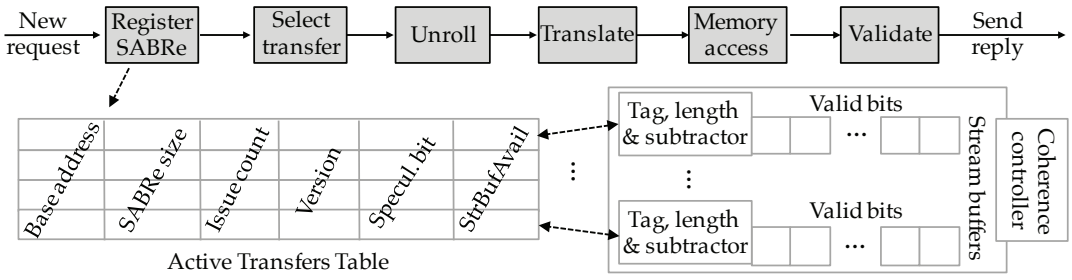


Figure 4: R2P2 microarchitecture.

driving a SABRe's progress is an SRAM structure, dubbed Active Transfers Table (ATT). An ATT entry represents a SABRe during its lifetime. Every ATT entry controls an associated stream buffer, and every stream buffer holds a base address, a length field, and a bitvector representing a range of consecutive cache blocks following the base address, with each bit representing a cache block. A set bit indicates that the cache block has been read from the memory subsystem. Each stream buffer also features a subtractor, used to determine whether a message from the memory hierarchy (reply or snoop) matches an entry in the stream buffer by subtracting the stream buffer's base address from the received address to index the bitvector. This simple lookup mechanism eliminates associative searches within each stream buffer.

Upon the reception of a new SABRe request, the *register SABRe* stage allocates a new entry in the ATT; the request carries the *SABRe size* and *base address*. The *select transfer* stage is a simple SABRe scheduler that selects one of the active SABRes in the ATT and starts unrolling it. The *unroll* stage issues load requests for the registered SABRe and increments the *issue count* while (i) *issue count* $<$ *SABRe size,* and (ii) there is a free slot in the associated stream buffer (*StrBufAvail*), *or* the object's version has already been read, so the SABRe is past its window of vulnerability (*speculation bit* cleared). If condition (ii) is not met, the serviced SABRe gets descheduled and the *select transfer* stage starts servicing another active SABRe.

For every reply that arrives to the R2P2, all stream buffers are snooped to check for an address match in their tracked address range; upon a match, the corresponding bit of the bitvector is set. A similar match is triggered by received invalidation messages; if the invalidated address matches a valid entry in a stream buffer (entry's bit set), the invalidation is propagated to the stream buffer's corresponding ATT entry. If the version for that SABRe hasn't yet been read (*speculation bit* set), this event implies a race condition with a writer, and therefore the SABRe aborts. Otherwise, if the version has already been read (*speculation bit* cleared), the invalidation is ignored, as it has to be triggered by a cache block's eviction from the chip.

The only ambiguous event is the reception of an invalidation for a stream buffer's base address, which represents the block that holds the target object's version. Such an invalidation message may be triggered by a real conflict from a writer concurrently writing the same object, or may be a false alarm triggered by the block's eviction from the chip. To avoid false conflicts, an invalidation for the SABRe's base address does not automatically abort the SABRe. Instead, we deploy the following mechanism: every cache block read from the memory hierarchy is directly sent back to the requester, and, after all payload replies for a SABRe have been sent back, the R2P2 sends a final payload-free packet indicating the transfer's atomicity success or failure. Whenever a SABRe's data accesses finish and the base address entry is still valid in the corresponding stream buffer, the R2P2 *immediately* confirms the SABRe's success. In the uncommon event of an invalidation reception for the SABRe's base block, the R2P2 must verify whether there was a true atomicity violation: after all data blocks for the SABRe have been read, the R2P2's *Validate* stage reads the object's header again and checks if the newly read version matches the ATT entry's *version* field (initialized the first time the object's header was read). A version match guarantees atomicity, while a mismatch implies atomicity violation and causes a SABRe abort.

The relative location of the lock/version in each object's header with respect to its base address is fixed for a given data store, but may vary across data stores. While LightSABRes require this information, a device driver can trivially specify that at initialization time, when it registers the data store's memory to the protocol controller, thus associating that metadata with the registered memory chunk.

5. LightSABRes on Scale-Out NUMA

We now integrate LightSABRes into soNUMA [34], a state-of-the-art rack-scale architecture, as a case study. We first provide a basic overview of soNUMA and its key characteristics affecting the integration.

soNUMA is a programming model, architecture, and communication protocol for low-latency, high-bandwidth in-memory computing and data serving. A soNUMA cluster consists of multiple SoC server nodes connected by an inter-node fabric. Nodes communicate via one-sided remote read and write operations, similar to RDMA. Remote accesses spanning multiple cache blocks are *unrolled into cache-block-sized requests* at the source node. Source unrolling was a conscious choice of key importance in the original soNUMA protocol design, as it facilitates transport-layer flow control by forcing a strict request-reply scheme.

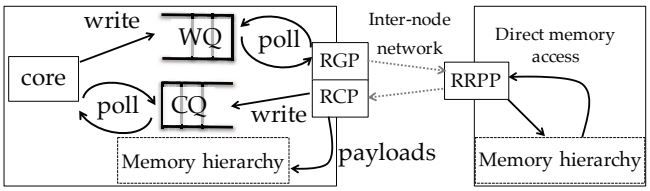


Figure 5: soNUMA remote read illustration.

soNUMA's protocol controller, the Remote Memory Controller (RMC), is directly integrated into the chip's coherence domain and handles the remote memory access requests scheduled by the cores. Fig. 5 shows the three distinct stages every request goes through: generated at the requesting node, serviced at a remote node, and completed once it returns to the requesting node. These three logical stages are handled by three independent pipelines. Request generations and completions are communicated between the cores and the RMC pipelines through memory-mapped and cacheable queues, a Work Queue for new requests and a Completion Queue for completed requests, as in RDMA. Since LightSABRes only involve destination-side processing, we only focus on the remote end's pipeline, namely the *Remote Request Processing Pipeline*, which statelessly services incoming remote requests by reading or writing local memory. A more detailed description of soNUMA and its pipelines can be found in [34].

5.1. Adaptation to soNUMA's requirements

The two key characteristics we need to take special care of in a LightSABRes implementation for soNUMA are the source unrolling of requests and the one-to-one request-reply invariant. At the high level, while soNUMA's Remote Request Processing Pipeline originally serves cache-block-sized requests in a stateless manner, SABRes inherently require some state: request packets belonging to the same SABRe are related. Now transformed into an R2P2, the pipeline gradually folds the received request packets belonging to the same SABRe into a single entry. For that purpose, we add two more fields to the ATT: the *SABRe id* and the *request counter*. A new SABRe is registered in the ATT by a special *SABRe registration* packet, with a *SABRe id* uniquely defined by the set of source node id, Request Generation Pipeline id, and transfer id, all of which are carried in each request packet. A registered SABRe's *request counter* is incremented for every consequent request packet belonging to the same SABRe (matching *SABRe id*). An additional limitation to the *unroll* stage is that requests to the memory hierarchy can be issued only if *issue count* $<$ *request counter* as well, to guarantee that the number of generated replies never exceeds the number of received requests.

Upon a SABRe abort, the R2P2 could transparently retry the failed SABRe. However, we consciously opt out of this approach for two reasons. First, retrying a failed SABRe in hardware will directly increase the occupancy of the R2P2 and also transparently increase the remote read's completion latency for an arbitrary amount of time from the application's perspective. Second, unless a conflict is detected on the first data block read, retrying a failed SABRe at the remote end will result in repeating some reply packets, thus breaking the request-reply flow control invariant of soNUMA. We choose to make the common case fast and expose the uncommon case of atomicity violation to software, to provide end-to-end control and flexibility. The application decides whether to retry an optimistic read after a backoff, or read the object over an RPC. Such policies are hard to implement solely in hardware, and the expected low abort rates do not justify the complexity and effort.

Properly sizing the ATT and the stream buffers, both in terms of number and depth, is key to the LightSABRes' performance. As detailed in §4.1, sizing is determined by the chip's memory hierarchy and the R2P2's target peak bandwidth. For our modeled 16-core system with an average memory access latency of 90ns and a target per-R2P2 peak bandwidth of 20GBps, we equip the LightSABRes with 16 stream buffers (one per ATT entry) and a depth (bitvector width) of 32, numbers simply derived by our target bandwidth-delay product (Little's Law). With 24 bytes per ATT entry and 11 bytes per stream buffer, the total additional per-R2P2 hardware requirement is 560 bytes of SRAM storage, plus a 42-bit subtractor per stream buffer.

Multicore chips may feature multiple R2P2s (e.g., [8]), which introduces a load-balancing concern for incoming SABRes. Since SABRes can be arbitrarily long and can differ in size, distributing a single SABRe across multiple R2P2s results in finer-grain load balancing. However, such distribution requires breaking a SABRe operation into multiple sub-operations that *all together* have to be atomic, introducing additional hardware design complexity. Furthermore, given that each transfer originates from a single Request Generation Pipeline, all reply packets have to be routed back to that pipeline's matching Request Completion Pipeline, which will ultimately become the transfer's bandwidth bottleneck. Therefore, the additional complexity required for inter-SABRe distribution seems unwarranted and our LightSABRes implementation for soNUMA maps each SABRe to a single R2P2.

5.2. soNUMA protocol & hardware extensions

Enhancing soNUMA with SABRe operations requires some modifications to the protocol and the remaining two RMC pipelines, namely the Request Generation and Request Completion Pipeline. The hardware-software interface is enhanced with a new SABRe operation type and an additional *success* field in the Completion Queue entry. This field is used by the Request Completion Pipeline in the Completion Queue entry to expose SABRe atomicity violations to the application. At the transport layer, we add two new packet types. The first is the *SABRe registration* packet, which precedes the SABRe's data request packets and contains the SABRe's total size; this is essential for the SABRe's

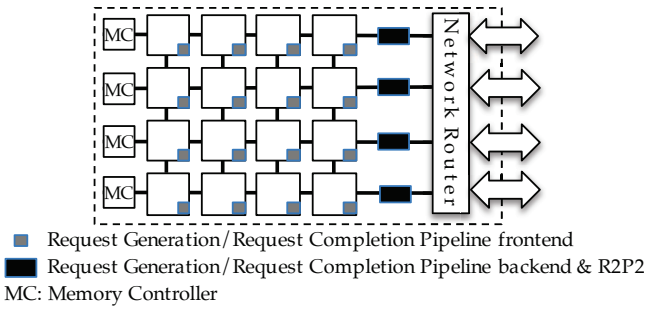


Figure 6: Multicore chip with multiple split RMCs.

registration at the destination node's R2P2 ATT. We assume a network that guarantees in-order packet delivery, but the mechanism can be easily extended to unordered networks, by carrying that information in every request packet. The second new packet type is the *SABRe validation*, which is the last reply sent by the R2P2 to indicate a SABRe's atomicity success or failure.

The Request Generation and Request Completion Pipelines need to comply with the aforementioned protocol changes. The former pipeline is extended to recognize the new SABRe request type and send a first *SABRe registration* packet to the destination before unrolling the data request packets. The latter pipeline is extended to recognize the *SABRe validation* packets carrying the success/failure information for a SABRe, and to encode the SABRe's success in the corresponding field of the Completion Queue entry upon reception of the SABRe's last reply packet.

6. Methodology

System organization. We evaluate LightSABRes by modeling two directly connected 16-core chips that implement soNUMA and have their RMCs enhanced with our proposed extensions. Fig. 6 shows the layout of the modeled chips, which implement a state-of-the-art manycore NI design. Each chip features multiple RMCs in a split-NI configuration [8]. Request Generation and Request Completion Pipelines (RGP and RCP) are split into frontends and backends; frontends are replicated per core and handle the memory-mapped queue-based interaction with the cores, while backends are replicated across the chip's edge, for efficient data handling. R2P2s are monolithic and replicated across the chip's edge.

Simulation. We use Flexus [47], a full-system cycle-accurate simulator, to evaluate our LightSABRes-enhanced soNUMA system. Table 2 summarizes the used parameters.

Applications. We use a simple microbenchmark to study the performance of LightSABRes in isolation. Our microbenchmark launches a number of writer threads that update objects in their local memory, or reader threads that access objects in remote memory using one-sided soNUMA operations (remote reads or SABRes) in a tight loop.

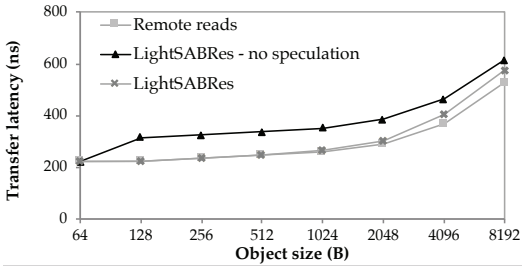
We also use FaRM [12] to evaluate the effect of LightSABRes on a full software stack. FaRM is a transactional system for distributed memory with an underlying key-value data store, that uses RDMA for fast remote memory access. In particular, FaRM uses one-sided reads to access remote objects over RDMA, while writes are always sent to the data owner over an RPC. FaRM implements atomic remote object reads via optimistic concurrency control by encoding per-cache-line versions in the objects. The framework detects atomicity violations for (local or remote) reads, should they overlap with a concurrent write to the same object, and retries the read operation. FaRM provides a fast path for lock-free single-object remote read operations, which are strictly serializable with FaRM's general distributed transactions, without invoking the distributed transactional commit protocol. As discussed extensively in §2.1, the per-cache-line versions mechanism imposes CPU overheads related to extracting the useful data from the data store and also requires intermediate system-managed buffering before exposing the data to the application, giving up on the zero-copy benefit that one-sided reads can provide.

We performed the following major modifications to FaRM: (i) we ported the FaRM core from a standard RDMA interface to soNUMA [34]; (ii) because of the current constraints of soNUMA and Flexus, we ported FaRM from Windows/x86 to Solaris/UltraSPARC III. We also replaced a number of system calls in FaRM, such as timer-related calls, with their most efficient counterparts on Solaris.

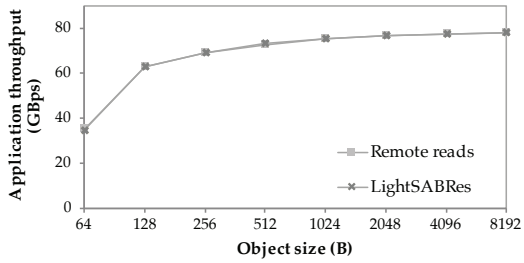
We evaluate two implementations of atomic lock-free reads with different object layouts in the FaRM data store. In the baseline implementation, we use soNUMA's remote read primitives combined with the original FaRM data object store (per-cache-line versions layout) and post-transfer atomicity checks in software. In the SABRe implementation, we remove these per-cache-line versions from the objects' layout and use LightSABRes to enforce atomicity. The SABRe implementation also removes the intermediate buffering for the data transferred from remote memory; instead, the one-sided operation can directly write the—already clean—data into the application buffer (zero-copy).

Cores	ARM Cortex-A57-like; 64-bit, 2GHz, OoO 3-wide dispatch/retirement, 128-entry ROB, TSO
L1 Caches	32KB 2-way L1d, 48KB 3-way L1i, 64-byte blocks 2 ports, 32 MSHRs, 3-cycle latency (tag+data)
LLC	Shared block-interleaved NUCA, 2MB total 16-way, 1 bank/tile, 6-cycle latency
Coherence	Directory-based Non-Inclusive MESI
Memory	50ns latency, 4x25.6GBps (DDR4)
Interconnect	2D mesh, 16B links, 3 cycles/hop
RMC	3 independent pipelines (RGP, RCP, R2P2) @ 1GHz one RGP/RCP frontend per core (Fig. 6) four RGP/RCP backends & R2P2s across edge
LightSABRes	16 32-entry stream buffers per R2P2
Network	Fixed 35ns latency per hop [42], 100GBps

TABLE 2: System parameters for simulation on Flexus.



(a) End-to-end latency.



(b) Bandwidth.

Figure 7: Microbenchmark with one-sided operations.

7. Evaluation

7.1. Latency and throughput characterization

We first use a single-threaded microbenchmark that issues synchronous operations, remote reads and SABRes, to assess their latency. To illustrate the benefit of the LightSABRes mechanism over a basic hardware mechanism for SABRes that serializes the version check before data access, we evaluate the performance of both mechanisms (*LightSABRes* vs. *LightSABRes - no speculation*). Remote data is memory resident and the local buffer at the source is LLC resident. Fig. 7a shows the soNUMA transfer latency (from issuing to completion) as a function of the transfer size. For single-block transfers, *remote reads* and both types of *LightSABRes* achieve the same latency, as expected. For larger transfers, the latency of SABRes using the *LightSABRes - no speculation* mechanism is significantly higher than *remote reads*, because of the read-version-then-data serialization. *LightSABRes* successfully remove this overhead, matching the latency of *remote reads*. The latency difference between *remote reads* and *LightSABRes* in the case of large transfers (>2KB) is attributed to the load distribution to R2P2s: while remote reads are balanced on a per-block basis across R2P2s, each SABRe is assigned to a single R2P2.

Fig. 7b shows the peak throughput of 16 threads issuing asynchronous remote operations (remote reads and SABRes). The remote reads and LightSABRes have identical throughput curves, illustrating that (i) peak theoretical bandwidth (20GBps per R2P2) is reached with both operation types, and (ii) introducing state at the R2P2s does not hurt throughput. The throughput curve of *LightSABRes - no spec.* is also identical, and therefore omitted.

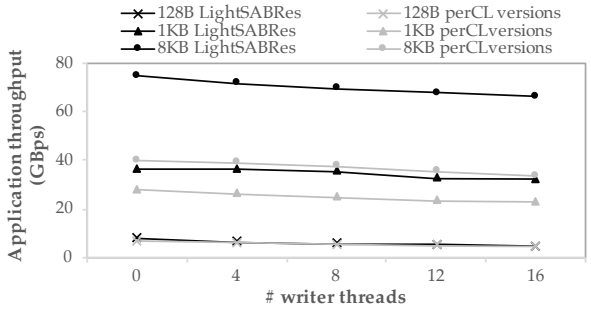


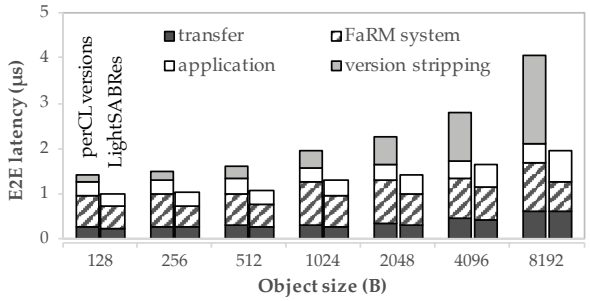
Figure 8: App. throughput with increasing conflict rate.

7.2. Conflict sensitivity

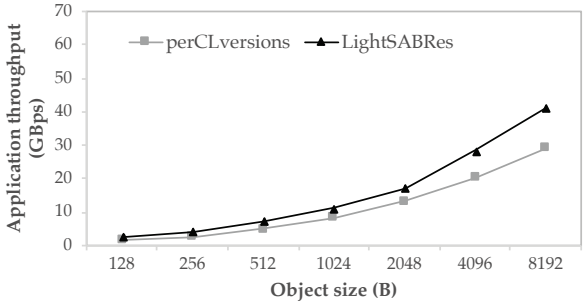
We extend the synchronous microbenchmark used in §7.1 to evaluate the end-to-end effect of LightSABRes in the presence of atomicity violations. We use the per-cache-line versions technique to provide atomicity in software, using remote reads. After every transfer, the microbenchmark unpacks the transferred data into an application buffer, checking for atomicity violation in the process. With LightSABRes, such an atomicity check mechanism is not required. In both cases, the end result is the same: a remote operation completes when the clean data is read by the core.

We employ 16 reader threads on one chip and vary the number of writers from 0 to 16 on the other, for a throughput sensitivity analysis as the conflict probability grows. To achieve a perceivable change in conflict probability, we limit the number of objects to 100, making all accesses LLC resident. Readers access all remote objects uniformly at random, while each writer repeatedly writes a predefined subset of the objects (Concurrent Reads Exclusive Writes model [25]). Upon a conflict detection, readers immediately retry reading the same object again.

Fig. 8 compares the microbenchmark's throughput for remote atomic reads of 128B, 1KB, and 8KB objects, when using the software per-cache-line versions mechanism versus LightSABRes. In all cases, we observe a performance degradation as the number of writers, and, hence, the conflict probability, increases. The throughput difference between the software and hardware atomicity enforcement method is a direct result of the reduced end-to-end latency delivered by LightSABRes. We observe an opposite trend for small and large objects. For 128B objects, the application throughput gap between LightSABRes and the software mechanism shrinks from 15% to 3% as the conflict probability increases. In contrast, for 1KB and 8KB objects, the throughput gap grows from 30% to 41%, and from 87% to 97%, respectively. The reason for these differences is two-fold. First, the benefit from removing the software atomicity check is proportional to the object size. Second, atomicity success or failure of completed SABRes is directly exposed to the application through the transfer's Completion Queue entry. This action is object size agnostic. In contrast, the cost of software atomicity detection grows with the object size. Therefore, the larger the object size and the conflict probability, the greater the benefit for LightSABRes.



(a) End-to-end latency breakdown.



(b) Application throughput.

Figure 9: FaRM KV store: baseline vs. LightSABRes.

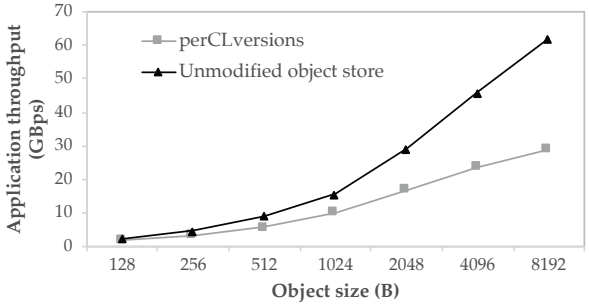


Figure 10: FaRM local reads throughput comparison.

7.3. FaRM

We conclude the evaluation by combining LightSABRes with a read-only key-value store application running on top of FaRM [12]. The first node allocates a number of FaRM objects in its memory, which a single reader thread running on the second node accesses continuously by issuing key-value lookups over synchronous one-sided operations: remote reads vs. SABRes. All remote memory accesses miss in the remote LLC and go to main memory.

Fig. 9a shows the latency breakdown for different object sizes and each of the two evaluated FaRM versions (baseline vs. SABRes). LightSABRes considerably reduce the end-to-end latency for atomic remote object reads for all object sizes. We identify two main sources of benefit. The direct benefit comes from the fact that SABRes completely remove the software overhead of version stripping and atomicity checking. The second, implicit, benefit is that SABRes shrink the total instruction footprint, thus reducing frontend stalls, which are critical to performance and a major concern in modern server workloads [13]. As pointed out in the methodology, SABRes not only deprecate the code for software atomicity checks, but also the FaRM code that deals with intermediate buffering, as SABRes allow soNUMA to directly write into the application buffer (zero-copy). We found the application's instruction working set to be in the 40-50KB range, which results in L1i conflict misses, even though we deploy a next-line instruction prefetcher. The use of SABRes reduces the instruction working set by ∼7%, relaxing core frontend pressure. The use of SABRes only increases the application's latency component (Fig. 9a), because the accessed object is located in the LLC, as opposed to the baseline where the software atomicity check implicitly brings the clean object in the L1d.

The application has two distinct phases: a low ILP/MLP phase with an IPC of 0.8 to 1, and a high-MLP phase, when the transferred remote data is read by the core. In the case of small objects, the largest fraction of the performance benefit provided by SABRes comes from the first phase. The combination of reduced instruction footprint (no version stripping or intermediate buffering code) and a slightly reduced instruction miss ratio results in a 35% overall latency improvement for 128B remote object accesses.

In contrast, the greatest benefit of SABRes for large objects comes from the high-MLP phase, increasing the performance benefit to 52% for 8KB objects. We do not model a data prefetcher, which would be capable of shrinking the gap between SABRes and the baseline for large objects. However, we significantly optimized the version stripping kernel by hand-tuning assembly code to maximize the MLP, at 1KB data chunks; thus, our results for object sizes up to 1KB are guaranteed to get maximum MLP, which a data prefetcher would not improve. Assuming a *perfect* data prefetcher that identifies the access to an object and directly brings all of it in the L1d, so that only the LLC access latency of accessing the first 1KB is exposed, the performance benefit of using SABRes would shrink from 52% to 30-35% for 8KB objects.

The latency benefit of using LightSABRes also results in throughput improvement. We now use 15 FaRM reader threads that access remote objects using synchronous remote operations (reads or SABRes). Fig. 9b shows that LightSABRes deliver a throughput improvement of 30-60% depending on the object size, as compared to the baseline.

Finally, we evaluate the performance of local reads for the two FaRM object store implementations. While LightSABRes are not involved in local accesses, they are an enabler for keeping the object store unmodified (i.e., no per-cache-line versions), which implicitly results in faster local reads. Fig. 10 shows the application throughput achieved for a read-only key-value lookup kernel on FaRM, with 15 FaRM reader threads issuing read requests to local memory only. We observe a throughput increase of 20% for 128B objects, which grows to 53% for 1KB objects, and a striking 2.1× for 8KB objects. Thus, using LightSABRes also results in a substantial acceleration of local reads, which are performance-critical even in distributed memory environ-

ments, especially in the case of locality-aware applications.

8. Related work

RPCs. In this work, we focused on one-sided operations. However, there is an important class of modern software frameworks, such as HERD [21] and RAMCloud [36], that still relies on RDMA for fast communication, but only uses it for fast messaging; all remote data is accessed over RPCs. While very flexible, RPCs forego the benefits of one-sided operations in terms of latency and remote CPU involvement. Hardware accelerators for one-sided operations, such as LightSABRes, can provide massive MLP, which is, in general, unattainable with RPCs, as their concurrency is fundamentally limited by the number of available cores.

In the broader sense of RPCs, our proposal for one-sided operations with stronger semantics (such as SABRes) and the addition of destination-side accelerators is semantically as much of an RPC mechanism as it is a one-sided operation. LightSABRes can be perceived as a simple fixed-functionality hardware RPC unit that reaps all the benefits of one-sided operations, and addresses the shortcomings of software RPCs at the price of limited flexibility. Our work is also parallel to a recent line of work on smart NICs [22], [23]. While we share the vision of adding more and smarter functionality to the network interface, we focus on providing operations with high-level semantics in hardware rather than more sophisticated network packet processing.

Hardware-software contract. The hardware simplicity of LightSABRes stems from the insight that objects in data stores are structured, and this software-provided guarantee can be harnessed. A similar observation has been made and leveraged before in the context of HTM: object-aware HTM relies on the organization of data as software objects to tackle the capacity limitations of traditional HTM [24].

Atomic chunk operations. A large body of work has been done in providing atomic access to memory chunks in shared-memory architectures [3], [4], [5], [10], [17], [37], [38], [46]. While these mechanisms can be used in a distributed memory environment to provide SABRes, they deliver broader functionality than simple atomic range reads at the cost of increased hardware complexity and intrusive hardware modifications. In contrast, LightSABRes only require simple and contained extensions to the integrated network protocol controller, without any further chip modifications (e.g., caches, cache and coherence controllers); thus, integration into commercial chips with conventional block-based coherence protocols is more practical.

Memory subsystem support. Tagged memory has been extensively investigated in the context of security and data integrity [7], [9], [41], [48]. Variations of such architectures can also be found on real machines, such as the Soviet Elbrus processors in the 70s [27], the J-machine in the 90s [40], and Oracle's recent M7 chip [1]. The hardware tags embedded in memory can be leveraged as a mechanism for concurrency control, e.g., as a hardware implementation of per-cache-line versions. The *destination-side* protocol controller could use these versions to identify atomicity violations while servicing a SABRe. While functionally similar to its software counterpart, such a hardware mechanism would be significantly more efficient, with the added benefit of leaving the data store's layout unmodified.

HICAMP [6] effectively provides snapshot isolation for all software objects through hardware multiversioning, thus preventing read-write conflicts. Integration of protocol controllers for one-sided operations with HICAMP is an interesting case where SABRes are provided by default, without any special hardware extensions.

9. Conclusion

The emergence of highly integrated rack-scale systems employing lightweight communication protocols and high-performance fabrics brings the remote memory access latency down to a bare minimum, within a small factor of local memory. In such systems, any software overheads added on top of the hardware latency for remote memory access are on the critical path and directly impact the end-to-end latency. This is the case for modern software mechanisms that provide atomic access to remote objects, which is a very common operation. We therefore introduced SABRes, a new one-sided operation that provides object atomicity in hardware. Our implementation, LightSABRes, completely removes the software overhead for atomicity enforcement, resulting in remote read throughput improvements of up to 97% for a microbenchmark and up to 60% for a key-value lookup application running on top of the full software stack of a modern distributed object store.

Acknowledgements

The authors thank the anonymous reviewers for their precious comments and feedback, as well as Dionisios Pnevmatikatos, Javier Picorel, Mario Paulo Drumond, and Rishabh Iyer for their feedback and suggestions. This work has been partially funded by the Nano-Tera *YINS* project, the CHIST-ERA *DIVIDEND* project, and the *Scale-Out NUMA* project of the Microsoft-EPFL Joint Research Center.

References

[1] K. Aingaran, S. Jairath, G. K. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, and T. Wicki, "M7: Oracle's Next-Generation Sparc Processor." *IEEE Micro*, vol. 35, no. 2, pp. 36–45, 2015.

[2] K. Asanović, "A Hardware Building Block for 2020 Warehouse-Scale Computers," USENIX FAST Keynote, 2014.

[3] C. Blundell, M. M. K. Martin, and T. F. Wenisch, "InvisiFence: performance-transparent memory ordering in conventional multiprocessors." in *ISCA*, 2009.

[4] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: bulk enforcement of sequential consistency." in *ISCA*, 2007.

[5] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A Scalable, Non-blocking Approach to Transactional Memory." in *HPCA*, 2007.

[6] D. R. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and O. Azizi, "HICAMP: architectural support for efficient concurrency-safe shared structured data access." in *ASPLOS-XVII*, 2012.

[7] J. R. Crandall and F. T. Chong, "Minos: Control Data Attack Prevention Orthogonal to Memory Model." in *MICRO*, 2004.

[8] A. Daglis, S. Novakovic, E. Bugnion, B. Falsafi, and B. Grot, "Manycore network interfaces for in-memory rack-scale computing." in *ISCA*, 2015.

[9] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security." in *ISCA*, 2007.

[10] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: deterministic shared memory multiprocessing." in *ASPLOS-XIV*, 2009.

[11] A. Dhodapkar, G. Lauterbach, S. Li, D. Mallick, J. Bauman, S. Kanthadai, T. Kuzuhara, G. S. M. Xu, and C. Zhang, "SeaMicro SM10000-64 Server: Building Datacenter Servers Using Cell Phone Chips," in *HOTCHIPS-XXIII*, 2011.

[12] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory." in *NSDI*, 2014.

[13] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware." in *ASPLOS-XVII*, 2012.

[14] M. Franklin and G. S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism." in *ISCA*, 1992.

[15] M. Franklin and G. S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," *IEEE Trans. Comput.*, 1996.

[16] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is SC + ILP=RC?" in *ISCA*, 1999.

[17] L. Hammond, B. D. Carlstrom, V. Wong, M. K. Chen, C. Kozyrakis, and K. Olukotun, "Transactional Coherence and Consistency: Simplifying Parallel Hardware and Software." *IEEE Micro*, vol. 24, no. 6, pp. 92–103, 2004.

[18] Hewlett-Packard Enterprise, "HP Moonshot System Family Guide," 2015. [Online]. Available: http://h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA4-6076ENW&cc=us&lc=en/.

[19] Hewlett-Packard Enterprise, "The Machine: A new kind of computer," 2015. [Online]. Available: http://www.labs.hpe.com/research/themachine/.

[20] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers." in *ISCA*, 1990.

[21] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services." in *SIGCOMM*, 2014.

[22] A. Kaufmann, S. Peter, T. E. Anderson, and A. Krishnamurthy, "FlexNIC: Rethinking Network DMA." in *HOTOS-XV*, 2015.

[23] A. Kaufmann, S. Peter, N. K. Sharma, T. E. Anderson, and A. Krishnamurthy, "High Performance Packet Processing with FlexNIC." in *ASPLOS-XXI*, 2016.

[24] B. Khan, M. Horsnell, I. Rogers, M. Luján, A. Dinn, and I. Watson, "An Object-Aware Hardware Transactional Memory System." in *HPCC*, 2008.

[25] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage." in *NSDI*, 2014.

[26] K. T. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing SoC accelerators for memcached." in *ISCA*, 2013.

[27] Linley Group, "The Russians are Coming," *Microprocessor Report*, February 1999.

[28] Linley Group, "X-Gene 2 Aims Above Microservers," *Microprocessor Report*, September 2014.

[29] Linley Group, "Oracle Shrink Sparc M7," *Microprocessor Report*, September 2015.

[30] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage." in *EUROSYS*, 2012.

[31] Mellanox Technologies, "Scale-out databases," 2016. [Online]. Available: http://www.mellanox.com/page/scale_out_database/.

[32] C. Mitchell, Y. Geng, and J. Li, "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store." in *USENIX ATC*, 2013.

[33] A. Muzahid, S. Qi, and J. Torrellas, "Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically." in *MICRO*, 2012.

[34] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA." in *ASPLOS-XIX*, 2014.

[35] Oracle, "Oracle Exadata Database Machine," 2016. [Online]. Available: http://www.oracle.com/technetwork/database/exadata/overview/index.html/.

[36] J. K. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. M. Rumble, R. Stutsman, and S. Yang, "The RAMCloud Storage System." *ACM Trans. Comput. Syst.*, vol. 33, no. 3, p. 7, 2015.

[37] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian, "Scalable and reliable communication for hardware transactional memory." in *PACT*, 2008.

[38] X. Qian, J. Torrellas, B. Sahelices, and D. Qian, "BulkCommit: scalable and fast commit of atomic blocks in a lazy multiprocessor environment." in *MICRO*, 2013.

[39] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, "Scalable Hardware Memory Disambiguation for High ILP Processors." in *MICRO*, 2003.

[40] E. Spertus, S. C. Goldstein, K. E. Schauser, T. von Eicken, D. E. Culler, and W. J. Dally, "Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5." in *ISCA*, 1993.

[41] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking." in *ASPLOS-XI*, 2004.

[42] B. Towles, J. P. Grossman, B. Greskamp, and D. E. Shaw, "Unifying on-chip and inter-node switching within the Anton 2 network." in *ISCA*, 2014.

[43] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases." in *SOSP*, 2013.

[44] Z. Wang, H. Qian, J. Li, and H. Chen, "Using restricted transactional memory to build a scalable in-memory database." in *EUROSYS*, 2014.

[45] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM." in *SOSP*, 2015.

[46] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors." in *ISCA*, 2007.

[47] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical Sampling of Computer System Simulation." *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.

[48] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware Enforcement of Application Security Policies Using Tagged Memory." in *OSDI*, 2008.