Efficient Distributed Memory Management with RDMA and Caching

Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal[†], Gang Chen[‡]
Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, Sheng Wang

National University of Singapore, [†]*University of California at Santa Barbara*, [‡]*Zhejiang University*

{caiqc,wentian,zhangh,ooibc,tankl,teoym,wangsh}@comp.nus.edu.sg
[†]agrawal@cs.ucsb.edu, [‡]cg@zju.edu.cn

ABSTRACT

Recent advancements in high-performance networking interconnect significantly narrow the performance gap between intra-node and inter-node communications, and open up opportunities for distributed memory platforms to enforce cache coherency among distributed nodes. To this end, we propose GAM, an efficient distributed in-memory platform that provides a directory-based cache coherence protocol over remote direct memory access (RDMA). GAM manages the free memory distributed among multiple nodes to provide a unified memory model, and supports a set of user-friendly APIs for memory operations. To remove writes from critical execution paths, GAM allows a write to be reordered with the following reads and writes, and hence enforces partial store order (PSO) memory consistency. A light-weight logging scheme is designed to provide fault tolerance in GAM. We further build a transaction engine and a distributed hash table (DHT) atop GAM to show the ease-of-use and applicability of the provided APIs. Finally, we conduct an extensive micro benchmark to evaluate the read/write/lock performance of GAM under various workloads, and a macro benchmark against the transaction engine and DHT. The results show the superior performance of GAM over existing distributed memory platforms.

PVLDB Reference Format:
Qingchao Cai, Wentian Guo, Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient Distributed Memory Management with RDMA and Caching. *PVLDB*, 11 (11): 1604-1617, 2018.
DOI: https://doi.org/10.14778/3236187.3236209

1. INTRODUCTION

Shared-nothing programming model has been widely used in distributed computing for its scalability. One popular example is distributed key-value store [6, 21, 33, 36, 44], which uses key-value APIs (e.g., `Put` and `Get`) to access remote data. In comparison, the shared-memory model that is able to accss remote data via memory semantics renders a unified global memory abstraction very attractive for distributed computing, since it not only unifies global data access, but also, more importantly, enables users to view distributed computing nodes as a powerful server with a single unified memory space and hence develop distributed applications in the same way as they do multi-threaded programming. In addition, the skewness in data access, which can cause overloaded nodes to be bottleneck in shared-nothing systems, can be gracefully handled in such a unified model by transparently redirecting access requests to less loaded nodes.

Proceedings of the VLDB Endowment, Vol. 11, No. 11

DOI: https://doi.org/10.14778/3236187.3236209

There have been many DSM (Distributed Shared Memory) systems [4, 7, 26, 40] proposed to combine physically distributed memory together to enforce a unified global memory model. These systems typically employ a cache to buffer remote memory accesses. To maintain a consistent view on cached data, they use synchronization primitives to propagate dirty writes and clear cached read, which incurs a significant overhead at synchronization points. In addition, requiring programmers to manually call the synchronization primitives to ensure data consistency makes it difficult to program and debug with the memory model.

The emergence of RDMA (Remote Direct Memory Access) further strengthens the attraction of a unified memory model by enabling network I/O as remote memory access. As shown in Table 1, the throughput of current InfiniBand RDMA technology (e.g., 200 Gb/s Mellanox ConnectX® -6 EN Adapter [31]) is almost approaching that of local memory access, and can be even better than NUMA (Non-Uniform Memory Access) inter-node communication channels (e.g., QPI [34]). Thus, several RDMA-based systems [14, 24, 30, 35] have been proposed to leverage RDMA to enable a unified memory abstraction from physically distributed memory nodes. However, they still require users to manually call synchronization primitives for cache consistency, and hence suffer from the same problems as the traditional DSM systems [4, 7, 26, 40].

A natural way to avoid the above problems is to simply abandon the cache such that each operation (e.g., `Read`/`Write`) is routed to the node where the requested data resides. However, even with RDMA, fine-grained remote memory access still incurs a prohibitively high latency. As shown in Table 1, while the throughput of recent RDMA technology is approaching that of local memory access, its latency still lags farther behind.

This paper presents GAM, which adopts an alternative approach to the unified global memory model by reserving the cache to exploit locality in data accesses and leveraging RDMA to employ an efficient cache coherence protocol to guarantee data consistence and hence facilitate programming and debugging. The contributions made in this paper are summarized below:

- We propose a distributed in-memory computing platform – GAM, based on RDMA. GAM manages the distributed memory to provide a unified global memory model, and provides a set of APIs for global memory operation. GAM employs PSO memory consistency by adopting a programming model

Table 1: Communication Performance Comparison [32, 19, 37, 38], where **QDR**, **FDR**, **EDR** and **HDR** respectively represent Quad, Fourteen, Enhanced and High Data Rate.

Metrics	Local Memory (4 channels DDR3-1600)	QPI	IB QDR		IB FDR		IB EDR		*IB HDR*		10GbE
			4×	12×	4×	12×	4×	12×	4×	12×	
Throughput (GB/s)	51.2	16	5	12	7	21	12.5	37.5	25	75	1.25
Latency (μs)	<0.1	0.2-0.3	1.3		0.7		0.5		<0.5		5-50

of synchronous reads and asynchronous writes. A set of distributed synchronization primitives, such as lock and memory fence, are provided to enforce stronger consistency.

- We add another level of distributed cache on top of the global memory to exploit data locality and hide the latency of remote memory accesses. An efficient distributed cache coherence protocol is also designed based on RDMA to enforce the memory consistency model provided by GAM. Various special features of RDMA (e.g., one-sided direct memory access, pure header notification, packet inlining) are exploited for an efficient protocol implementation. In addition, we also design a logging scheme to support failure recovery.
- We build two applications: a transaction engine and a distributed hash table (DHT), by using the APIs provided by GAM, to demonstrate how GAM can be used for building high-level applications.
- We conduct an extensive micro benchmark to profile the performance of GAM from various perspectives, and a macro benchmark to show the superior performance of GAM over L-Store [27], FaRM [14], Tell [30], Grappa [35] and Argo [24] in terms of distributed transaction processing and DHT. The results illustrate the feasibility of implementing an efficient global memory model on modern RDMA network.

The rest of the paper is structured as follows. Sections 2 and 3 respectively present the system design and the RDMA-based implementation and optimization of GAM. Section 4 discusses the programming model and memory consistency of GAM. A logging scheme is designed in Section 5 to support failure recovery in GAM. Section 6 presents two applications delivered using GAM APIs. A performance study of GAM is given in Section 7. We present related work in Section 8, and finally conclude the paper in Section 9.

2. SYSTEM DESIGN

In this section, we introduce the system architecture of GAM. Based on the partitioned global addressing model, GAM provides a set of APIs for memory operations, and maintains cache coherence among distributed nodes to exploit data locality in applications.

2.1 Addressing Model and APIs

GAM adopts the partitioned global address space (PGAS) addressing model [10, 48], which provides a logically unified address space and hence simplifies the development of distributed applications by enabling them to be implemented as multi-threaded programs. At the hardware level, PGAS is realized by memories of many machines interconnected by RDMA network such that each node is responsible for one partition of the global address space.

A set of APIs listed in Table 2 are provided in GAM for the manipulation of the global memory space. These APIs can be classified into two categories: global memory access, which contains the first four APIs: `Malloc`/`Free` and `Read`/`Write`, and synchronization, to which the remainder APIs belong. We shall only discuss the APIs of the first category in this section, and defer the discussion of synchronization APIs to Section 4.

`Malloc`/`Free` is analogous to `malloc`/`free` in the standard library, except that it manipulates global memory instead of local

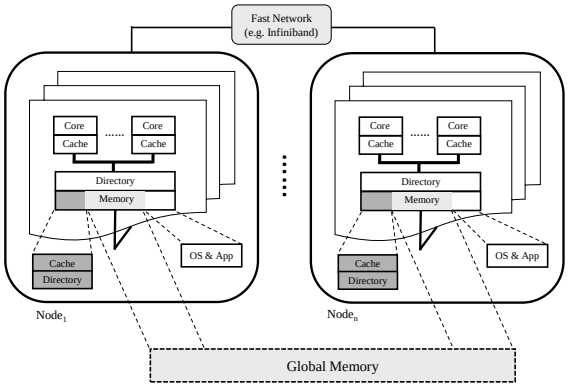


Figure 1: Overview of GAM

memory. If a base global address *gaddr* is provided to `Malloc` by the user as a hint of affinity, we shall allocate the memory in the same node with *gaddr*. Alternatively, users can also give some hints (e.g., local/remote) on the allocation, which will guide GAM to do the allocation accordingly (e.g., locally/remotely). By default, we first try to allocate locally; however, if the local node faces memory pressure, we then forward the allocation request to the node that has the most free memory, based on the global memory allocation statistics that are synchronized periodically. As we can see, GAM APIs are self-explanatory, and straightforward to use for building high-level applications on top of GAM.

2.2 Cache Coherence Protocol

Although the throughput and latency of RDMA networking have improved significantly, almost approaching those of QPI interconnect, there is still around 10× gap between local memory access and remote memory access. Given the locality property of most applications [13], the best practice to relieve the latency gap is to rely on the hierarchical memory architecture, i.e., multiple layers of caches/memory, to reduce the trips to the lower memory layers. In this regard, we add an extra level of DRAM resident cache atop the global memory to absorb remote memory accesses.

We decide not to use the snoop-based cache coherence protocol since broadcasting in RDMA network is unreliable, and building reliable broadcasting is costly and can easily overwhelm the network. Thus, we rely on the directory-based protocol by maintaining meta-data to keep track of data distribution in the cluster. In this way, we can have full knowledge of the data cached in each node, and keep the cache consistent by point-to-point communication.

The extra level of cache atop the global memory gives rise to three levels of cache coherence protocol in GAM as shown in Figure 1, i.e., the snoop-based protocol within a NUMA node, the directory-based protocol across NUMA nodes, and the distributed directory-based protocol we design. As the upper two levels of cache coherence are already enforced by hardware, we only focus on the lowest level of cache coherence protocol, which achieves a consistent view on global memory.

For each piece of data, there are five types of nodes: *home/remote node*, *request node*, *sharing/owner node*, depending on data location and access permission. The *home node* is the node where the

Table 2: GAM APIs

API	Input	Description
`Malloc/Free`	*size*, *gaddr*, *flag*	allocate/free memory in the global space, with affinity to *gaddr* and hints from *flag* (e.g., remote/local)
`Read/Write`	*gaddr*, *size*, *buf*	read/write the data from/to [gaddr, gaddr+size) to/from the local buffer
`Atomic`	*gaddr*, *func*, *args*	perform atomic function *func* on the global address *gaddr*
`MFence`	-	issue mfence synchronization
`RLock/WLock`	*gaddr*, *size*	lock (shared/exclusive) the global address [gaddr, gaddr+size)
`TryRLock/TryWLock`	*gaddr*, *size*	try-lock versions of RLock/WLock
`UnLock`	*gaddr*, *size*	unlock the global address

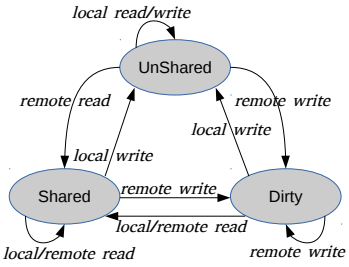


Figure 2: Transition between cache directory states

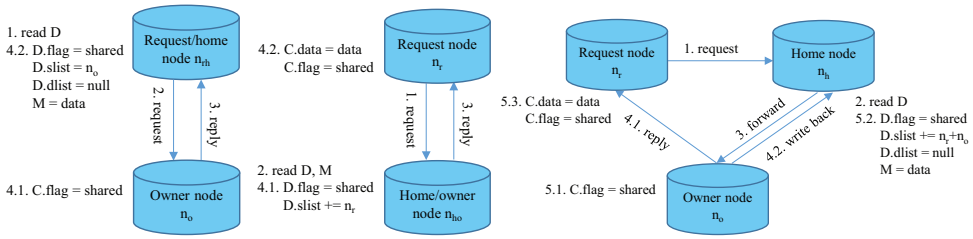


Figure 3: Workflows of Read Protocol (**D**, **C** and **M** stand for **D**irectory entry, **C**ache line and **M**emory line)

physical memory of the data resides, and all other nodes are *remote nodes*. The *request node* is the node requesting share/exclusive (i.e., read/write) access to the data. The *sharing/owner node* is the node that has share/exclusive permission of the data. For the sake of expression, we also say the owner node owns the data.

Initially, the home node owns the data, and is hence both the sharing node and the owner node. Upon receiving an access request for the data, the home node grants the respective permission to the request node, which is in turn promoted to either a sharing node or the owner node. Each data can be shared by multiple sharing nodes simultaneously, but has at most one owner node at a time. In addition, the owner node and the sharing nodes cannot co-exist, unless the owner node is the only sharing node. Data locality can be exploited when the request node has already been granted with the respective access permission.

Similar to the caching mechanisms in hardware, we also adopt the granularity of a cache line[1] to exploit the *locality* of data access, except that the cache line used in the distributed cache is configurable and much larger than the hardware cache to alleviate the high transmission cost for small packets[2].

The directory state of a cache line on the home node can be "Shared" (shared by some remote nodes that have read permission), "Dirty" (owned by a remote node that has write permission) and "UnShared" (owned by the home node). The transition between these three states is shown in Figure 2. Similarly, the cache state of a cache line on a remote node can be "Shared" (read-only), "Dirty" (writable) and "Invalid" (invalidated), depending on the data access permission held by that remote node. In addition, due to network delay, the transitions between states are not atomic, and thus we introduce an in-transition state for each possible state transition (e.g., "SharedToDirty"). As we shall see in Section 2.5, in-transition states are necessary to ensure the correctness in the processing of concurrent requests.

[1]Unless otherwise specified, "cache line" mentioned in this paper is by default referred to our software cache line.

[2]We are using 512 bytes as the default cache line size since it achieves a good balance between network bandwidth and latency.

2.3 Read

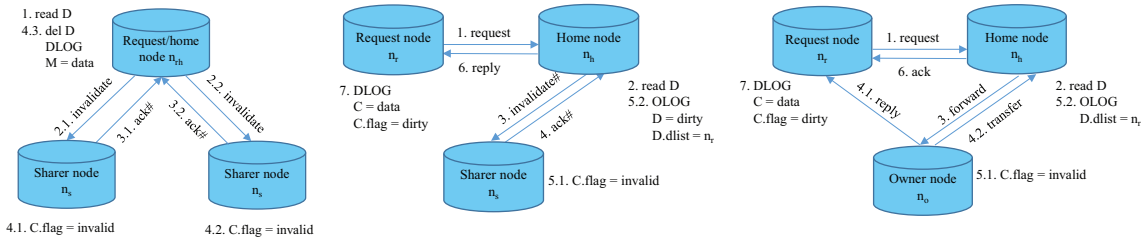
There are two types of reads in GAM, depending on the location of the request node. We term the reads whose request node is also the home node as local reads, and other reads as remote reads. In this section, we detail the workflow of both types of reads separately. Figure 3 gives a graphical illustration of read workflow. Each action of the workflow is associated with a major number and an optional minor number such that an action must be executed and completed before those with a higher major number, while two actions with the same major number can be executed simultaneously by different nodes.

2.3.1 Local Read

When a local `Read` is issued, if there is no remote node holding the ownership, the data either only resides in local memory ("UnShared") or is in read-only mode ("Shared"). In both cases, the data is supplied directly from local memory, without incurring any network communication. However, if there is a remote node owning the data ("Dirty"), the workflow becomes much more complex as it involves data transmission and cache state transition, both of which require inter-node communication. The detailed communication workflow is illustrated in Figure 3a. The home node n_h, which is also the request node n_r, first reads the corresponding directory entry (**D**) to obtain the information about the owner node n_o (1) and then sends a READ request to n_o (2), which in turn responds back with its own copy (3) and changes the state of the respective cache line (**C**) to "Shared" (4.1). Upon receiving the data, the home node n_h will update its memory (**M**) and the directory entry accordingly by changing the flag to "Shared"(4.2).

2.3.2 Remote Read

For a remote `READ` where the request node is different from the home node, if the requested data is already cached by the request node, then the cached copy can be directly used to service this read request without incurring any communication. Otherwise, depending on the cache directory state, there are two different workflows, which respectively correspond to the "UnShared/Shared" case (i.e. Non-Dirty) and the "Dirty" case.



(a) local, D.flag=shared **(b)** remote, D.flag=shared **(c)** remote, D.flag=dirty

Figure 4: Workflows of Write Protocol.(**D**, **C** and **M** stand for **D**irectory entry, **C**ache line and **M**emory line)

Figure 3b shows the situation where the requested data is "UnShared", and hence the home node is also the owner node. The request node n_r first notifies the home node n_h of the read request (1). After reading the cache directory (2) and finding itself holding the latest version of the requested data, the home node replies n_r with that data (3), and then accordingly updates the cache directory and share list (4.1). Finally, the request node n_r updates its cache with the received data (4.2). For the "Shared" case, although there are additional sharing nodes, the workflow is exactly the same as Figure 3b, as those sharing nodes do not need to participate.

In the second case where the requested data is owned by a remote node, the state of the respective cache line needs to be changed from "Dirty" to "Shared". As shown in Figure 3c, after realizing the requested data is dirty, the home node n_h needs to forward the read request to the owner node n_o (3). n_o will then send a copy of the cache line to both the request node n_r (4.1) and the home node n_h (4.2), and finally mark that cache line as "Shared" (5.1). Upon receiving the cache line copy, n_h will update the cache directory and write that copy into its memory (5.2), and n_r will put the copy in its cache in the hope of absorbing further read requests (5.3).

2.4 Write

Likewise, we also divide write requests into two categories: local writes and remote writes, and illustrate the respective workflow in Figure 4.

2.4.1 Local Write

For a write request issued by the home node, if there is no sharing or owner node, the home node can safely write the data without incurring any communication. Otherwise, the data is either shared ("Shared") or owned by a remote node ("Dirty"). In the first case, as shown in Figure 4a, the request node n_r, which is also the home node n_h, will issue an INVALIDATE request for each sharing node (2). Upon receiving the INVALIDATE request, each sharing node will acknowledge this request (3) and then invalidate its local copy (4.1). The acknowledgement (3) is necessary because it allows n_r to decide when to pass the memory fence, which will be discussed in Section 4. After collecting all acknowledgements from the sharing nodes, n_h will remove the directory entry of that data, and write new data to memory (4.2). In order to support failure recovery, n_h will also perform a DLOG to log the new data before it is actually written. The second case is similar to the first one, where the request node (i.e., the home node) will also instruct the owner node to invalidate the cache line and wait for the acknowledgement. The only difference is that the owner node need to piggyback the most recent cache line copy along with the acknowledgement in order for the request node to update its local memory.

2.4.2 Remote Write

A remote `Write` operation can be immediately fulfilled if the request node is also the owner node. Otherwise, there are three cases for the workflow, depending on the directory state of the cache line, which can be "Shared", "UnShared" or "Dirty".

The workflow of the first case is shown in Figure 4b. Upon receiving a write request from the request node n_r (1), the home node n_h checks its directory to obtain information about the sharing nodes(2), and then invalidates the cached copies on the sharing nodes (3). After receiving acknowledgements from all the sharing nodes (4), n_h performs an OLOG to log the ownership transfer of the respective cache line and updates the cache directory accordingly (5.2), after which an up-to-date version of the data is returned to the request node n_r (6). The second case, where the directory state of the cache line is "UnShared", is similar to the first case. The difference is that the home node n_h can now skip the step 3 and 4 in Figure 4b and grant the write permission to the request node immediately, since there are no sharing nodes.

Figure 4c illustrates the last case where the cache directory is "Dirty". After receiving the request from the request node n_r (1), the home node n_h first checks the cache directory to find the owner node n_o (2) and then forwards the request to n_o (3). Then, n_o sends the respective cache line to n_r (4.1), acknowledges the ownership transfer to n_h (4.2), and invalidates its local copy of that cache line (5.1). After receiving the ownership transfer message from the owner node n_o (4.2), the home node n_h is now able to grant the ownership to the request node n_r (6). But before that, it first needs to log that ownership by performing an OLOG, and update the dirty list as well (5.2). The acknowledgements from both n_o (4.2) and n_h (6) are necessary, because we have to keep the directory of the home node updated via the ownership transfer message (4.2) even in some scenarios (e.g., `TryLock`) where the forward request (3) is denied. After receiving both the reply from the owner node (4.1) and the acknowledgement from the home node (6), the request node n_r can now proceed by logging the new data, updating the cache line and setting its state as "Dirty" (7). It should be noted the operations of (7) can be performed only after both messages, i.e., (4.1) and (6), have been received. Otherwise, if the request node n_r were to perform (7) immediately after receiving (4.1), it may give up the ownership (e.g., due to an eviction) before the home node n_h receives the ownership transfer message from the owner node n_o (4.2), in which case the home node will believe the request node gets the ownership when in fact it no longer has the ownership.

2.5 Race Condition

Races can happen during the work flow discussed in the previous two subsections. For example, when a remote `Read`/`Write` operation is submitted and waiting for its reply, another remote `Read`/`Write` operation targeted at the same address is performed from another thread in the same node. This causes repeated requests and wasted network bandwidth. We avoid such situations such that during the processing of a request, each involved node considers the requested cache line is in an in-transition state, and blocks the processing of subsequent requests for the same cache

line until the current request is fulfilled. Hence, the in-transition states help to guarantee the atomicity of each primitive operation (e.g., read and write) within a cache line,

However, the in-transition states can cause deadlock. Suppose a request node with read permission wants to be promoted to write permission by issuing a WRITE_PERMISSION_ONLY message to the home node, while, in the mean time, the home node also wants to write to the same data and thus sends INVALIDATE message to this request node. In this case, both nodes need to wait for each other to acknowledge the respective message, which, as mentioned above, will be blocked due to the in-transition state of the requested cache line. To handle it, we require the request node to back off as if it was in the previous state, and leave the home node to resolve the inconsistency. Specifically, the home node will first set the corresponding directory as "UnShared" when issuing the INVALIDATE request. Upon receiving the WRITE_PERMISSION_ONLY message, it will be aware of the inconsistency caused by the back-off strategy and thus handle this request as if it were a WRITE request.

2.6 LRU-based Caching

GAM adopts the least recently used (LRU) cache replacement policy to manage the software cache. Each node maintains a local hashtable that maps global memory addresses to the respective cache lines. With more and more global memory accesses, new cache lines keep being added to the hashtable. Once the hashtable size exceeds a predefined threshold, the cache replacement module chooses a least recently used cache line to evict.

Since using only a single LRU list can incur a huge overhead when multiple threads are concurrently updating the list, we optimize for this case by introducing multiple LRU lists to trade the LRU accuracy for better performance. For each global memory access, the parallel threads randomly choose one LRU list to update; for each cache line eviction, one LRU list is randomly chosen to guide the cache line replacement. The eviction workflow depends on the cache line state, which can be either "Shared" or "Dirty". For the eviction of a "Shared" cache line, an eviction request is sent directly to the home node, after which the cache line can be re-allocated for other data immediately. For the eviction of a "Dirty" cache line, the owner node attaches a cache line copy to the eviction request, and waits for an acknowledgement from the home node before erasing the cache line. Thus, that cache line can still be used to service requests issued before receiving the acknowledgement.

2.7 Discussion

Although GAM significantly augments the memory area that can be accessed by applications, it may still be the case that the total memory consumption exceeds the global memory offered in GAM. We leave this out-of-global-memory case as a future work of GAM, and explore how recent advances in RDMA, such as ODP (On-Demand Paging), can be leveraged for handling it.

3. RDMA-BASED IMPLEMENTATION

In this section, we describe how we take advantage of RDMA networking to realize the cache coherence protocol described in Section 2.2. The architecture of GAM is shown in Figure 5, where the main module is the GAM coordinator, which implements the cache coherence protocol, and is used to coordinate local and remote memory accesses. There is one GAM master in the whole system, which is only used for initialization and status tracking. A unified memory model is exposed to upper-layer applications, which can access the entire memory in the cluster without knowing the actual location of the data.

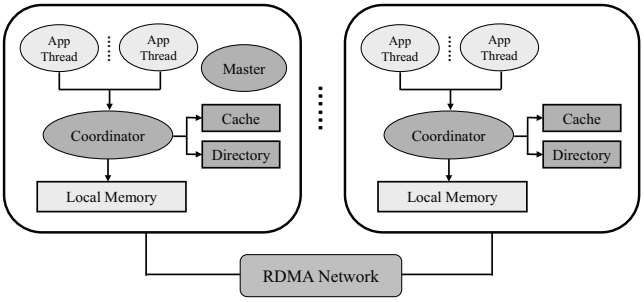


Figure 5: Architecture of GAM

3.1 Protocol Implementation

RDMA provides an efficient data transmission mechanism that can bypass the CPU and OS to access remote memory directly. However, a completely different transmission model (e.g., different APIs, one-sided transmission) and explicit communication buffer management, make it nontrivial to be employed effectively and efficiently. Basically, there are two sets of RDMA *verbs* that can be used to initiate data transmission, i.e., `READ` and `WRITE`, and `SEND` and `RECEIVE`. The `READ`/`WRITE` *verbs* are one-sided operations where only the requester is involved. The `SEND` *verb*, however, requires receiver side to post a `RECEIVE` *verb* to its *receive queue* beforehand. Generally, the one-sided `READ` and `WRITE` *verbs* perform better than the two-sided `SEND`/`RECEIVE` versions. since the former involves no CPU or OS at the receiver side, while the latter requires the receiver side to post `RECEIVE` *verbs* beforehand, and get notified after receiving the messages. However, for one-sided *verbs*, since the receiver does not get any notification, it is difficult to figure out the completion of a data transmission.

To utilize the features of RDMA for various needs, we design separate communication channels for control messages and data transmission, respectively. The rationale behind is that control messages (e.g., `READ`/`WRITE` requests) are relatively small and require instantaneous notification to the receiver, while data is transmitted in large units (i.e., cache line). For control channels, we avoid using RDMA `WRITE` *verb* and busy memory polling as in [14, 21], but resort to two-sided RDMA `SEND`/`RECEIVE` *verbs*. This is because, busy polling will consume huge amount of CPU resources in contrast to event-based mechanisms (e.g., epoll and select), and per-pair (sender/receiver) communication buffer would take up much memory. In addition, our communication does not always follow the request/reply pattern like key-value store, and hence the updates on the communication buffer ring at the receiver are difficult to be piggybacked to the sender. However, for the data transmission with larger transmission volume, we adopt one-sided RDMA `WRITE` to construct data channels. It allows to write directly to the final destination address, which is different from the usage of a dedicated communication buffer [14, 21] that always requires additional data copy between local memory and the registered communication buffer.

In addition, a special notification channel is implemented by using the RDMA `WRITE_WITH_IMM` with/without payload. For pure notification communication, only the request identifier is embedded into the header as the immediate value (32 bits) without any payload, which is more efficient in both requester and receiver sides [23]. If payload is also needed, the data channel and the notification channel will be combined together such that the receiver will get notified upon receiving the data. In most cases, the data channel is combined with the notification channel to achieve both large data transmission and efficient notification. We do not use `READ` *verb*

as it is known to have worse performance than WRITE [21], and it is difficult to guarantee data consistency in an efficient way [14]. For all communication channels, We use **RC** (reliable connection) transport type, since we require reliable transmission and strict orderings of the messages.

The workflows described in Sections 2.3 and 2.4 are implemented with the above three communication channels. The short control messages, i.e., "request", "forward" and "invalidate" messages, are transferred over the control channel (RDMA SEND/RECEIVE), while the "reply" and "writeback" messages are delivered over the combined data and notification channel (RDMA WRITE_WITH_IMM with payload). In addition, since the acknowledgements (e.g., "ack" and "transfer") only deliver success information, the notification channel solely (RDMA WRITE_WITH_IMM without payload) suffices for this purpose. Error replies still require the control channel since the requester needs more feedback information than a 32-bit immediate value.

3.2 Optimizations

We introduce below optimizations which exploit special features of RDMA to further improve GAM's performance.

Due to the limited size of the on-NIC cache, it is important to keep, as small as possible, the data needed by RDMA NIC, which typically include the *page table*, *queue pairs* and *receive queues*. To that end, we first organize the memory exposed for remote access as huge pages to reduce page table entries and in turn TLB (translation lookaside buffer) misses. In addition, all threads within one node share the same set of RDMA connections (i.e., *queue pairs*), which reduces the total number of *queue pairs* to n^2 (where n is the number of nodes in the cluster) in contrast to $n^2 \times t^2$ (t is the number of threads in each node), if every thread is connected directly with each other. This reduction in the number of *queue pairs* does not impair the throughput. As we have measured, one *queue pair* for each node suffices to saturate the RDMA NIC. Furthermore, for each node, we use only one *shared receive queue* which is shared among all associated *queue pairs*. This way, we not only reduce the state information that the NIC has to maintain, but also reduce the RDMA RECEIVE *verbs* that need to be posted beforehand.

To reduce the CPU overhead, selective signaling technique is used for the notification of completion. In particular, RDMA *verbs* are signaled every r requests, reducing the number of completion notifications and cleanup routines (e.g., free the send buffer) by $r\times$, which, as tested, leads to a significant performance improvement. To further reduce the network communication overhead, we use RDMA inline technique to send small payload directly via PIO (Programmed Input/Output) whenever possible to eliminate the extra DMA round trip via PCIe. The buffers used for inlined requests do not need to be registered beforehand, and can be re-used immediately after posting. This especially benefits the control channel, since its payload size is usually small enough to be inlined.

To exploit sharing opportunities among requests, we merge multiple small packets into one large packet while keeping the strict packet ordering. What is more, as mentioned in Section 2.5, in order not to send duplicated requests, we add all requests for the same cache line to a pending list, and process them one-by-one in the order they were issued.

4. MEMORY CONSISTENCY MODEL

There are a wide spectrum of memory consistency models, ranging from the strong consistency models (e.g., strict consistency and sequential consistency) to some relaxed ones (e.g., total store order (TSO), partial store order (PSO) and release consistency). Basically, the consistency models depend on the degree to which the order of global memory accesses is relaxed. Since there are only two types of memory access: Read and Write, there are totally four memory access orderings[3]: Read-After-Read, Read-After-Write, Write-After-Read, Write-After-Write, and relaxing different ordering leads to different memory consistency. For example, relaxing Read-After-Write to allow reads to return earlier than older writes leads to TSO, and further relaxing Write-After-Write results in PSO. Allowing reordering all four memory access orders yields release consistency.

Stronger consistency makes it easier to reason about memory access and decrease the programming complexity and debugging difficulty. Ideally, we can minimize the burden on users in programming with the unified memory model by enforcing strong consistency, such as sequential consistency or even strict consistency. However, although RDMA has already made network latency much less of a concern than before, enforcing strong consistency still incurs an unaffordable remote memory access latency as it requires both reads and writes to be performed synchronously. We hence first relax the Read-After-Write ordering to allow asynchronous Write and remove the Write from the critical path of program execution.

Our next decision is to further relax the Write-After-Write ordering. If we were to keep this ordering and hence provide TSO consistency, we need to issue the background write requests such that each write request can be issued only after all earlier requests have been completed, which means that the request node has been granted with all necessary write permissions. This substantially reduces the opportunity of write request coalescing and in turn incurs higher network overhead. Moreover, in this case, an overloaded node would block the following write requests to other nodes and slow down the entire system. As such, we relax the Write-After-Write ordering to permit a write request to be made visible before the earlier write requests.

Although the relaxation on the Read-After-Write and Write-After-Write ordering also relaxes the memory consistency of GAM, the programming complexity and program correctness do not get much affected as most programmers are familiar with the programming model of asynchronous writes (e.g., file IO), and the correctness of most programs does not rely on writes. Further relaxing the other two ordering, however, would result in a completely asynchronous programming model, which significantly increases the complexity and difficulty of programming.

Therefore, GAM provides PSO consistency by relaxing the Read-After-Write and Write-After-Write ordering, and hence employs a programming model of synchronous read and asynchronous write that most programmers are familiar with. To do so, a worker thread is dedicated to handling all requests from application threads. After issuing a Read request, the application thread will be blocked until the worker thread has fetched requested data into the given buffer. On the other hand, after issuing a Write request, the application thread immediately returns without waiting for that request to complete. Stronger consistency can be easily enforced, by using the explicit synchronization primitives (e.g., MFence, Lock/UnLock) that will be discussed in the next section. For example, sequential consistency can be easily achieved by inserting MFence following each Write operation. Lock primitives also facilitate the realization of application-level serializability (e.g., transactional serializability achieved by our transaction engine illustrated in Section 7.2).

[3]In the context of GAM, the memory access order is exactly the program order that the Read/Write APIs are called; the compiler will not reorder these calls as GAM is provided as a library which is linked to upper-level applications during runtime.

4.1 Synchronization Operations

In order to achieve a stronger consistency for higher-level applications, we provide two sets of explicit synchronization operations – *memory fence* (i.e., `MFence`), an operation analogous to the `MFence` operation in x86 instruction set, and *distributed* `Lock` *operations*, including shared lock, exclusive lock and their try-lock variants. A `Lock` operation is an implicit memory fence. For each `Lock` and `MFence` operation, its following operations will be pended until all previous operations have been completed by the worker thread. In addition, GAM also provides `Atomic` operations, which perform a given function against a global address atomically. For each `Atomic` operation, the worker thread considers it as an implicit write operation to the given address and hence tries to grasp the respective write permission. Thereafter, the worker thread calls the given function against that address.

`Lock` operations provide a natural synchronization mechanism for programmers to coordinate the data accesses in the shared memory environment. In our current design, lock and data are coupled together, and distributed locks across nodes hence have the same granularity as data sharing, i.e., a cache line, meaning that data in the same cache line shares the same lock. A `Lock` operation prefetches the requested cache lines into local cache and makes the request node become either a sharing node (in the case of `RLock`) or the owner node (in the case of `WLock`), making subsequent `Read`/`Write` or `Unlock` operations on the same cache lines free from extra network communication. Similarly, if the request node has already been granted with appropriate permission for the requested cache lines, the `Lock` operation can then be serviced locally. In this sense, `Lock` operations, like their `Read`/`Write` counterparts, can also be cached.

In addition, blocking `Lock` operations are implemented in a queue-based manner rather than repeated requesting over RDMA, which may overflow the network. By queue-based locking scheme, after the `Unlock` operation, we will choose the head node of the lock-waiting queue to grant the lock permission. However, `TryLock` operations are not queued; instead lock failure will be returned immediately if others are holding mutually exclusive locks. Some undo procedure and special handling are needed upon lock failure in order to maintain a consistent state, as it may involve multiple nodes, and it is possible that only one node rejects to grant the lock while others agree. We omit the description of the communication workflows for the shared or exclusive `Lock` operations, as they are similar to their corresponding `Read` (Figure 3) or `Write` (Figure 4) counterparts, respectively, except that some lock semantics are needed after acquiring the data.

Notice that the data in the same cache line share the same lock across nodes, so the blocking `Lock` operations may cause unexpected deadlock in, for example, some complex transaction processing applications. So non-blocking lock primitives are recommended in these applications, unless it is guaranteed to follow a certain order in the lock acquisition phase in all the processes. We intend to provide a fine-grained locking mechanism by decoupling the lock and data in the future.

5. LOGGING AND FAILURE RECOVERY

In this section, we design a logging scheme for GAM and show how it can be used for failure recovery.

There are two types of logging: DLOG and OLOG, which respectively log data writes and ownership transfers. As shown in Figure 4, DLOG is called by request nodes before writing data to memory/cache, and OLOG is called by home nodes before each ownership transfer. In addition, when acquiring an exclusive lock, the request node also calls DLOG to log the cache line which is prefetched along with the lock acquisition. We avoid logging data writes resulted from read requests which simply copy a cache line from the owner node; this helps reduce logging overhead, and does not affect failure recovery, as shown below. In addition, each log entry also includes a counter which is incremented upon each ownership transfer. In order to minimize the performance degradation incurred by logging, we rely on a NVRAM to accommodate the in-memory log, and asynchronously spill the content to SSDs/hard disks when the log is about to be full.

We shall now explain how the log can be used for failure recovery. We first consider the single failed node case, and then discuss the case of multiple failed nodes. For the sake of discussion, we assume each data write overwrites an entire cache line, and hence each DLOG entry also logs an entire cache line, which allows to recover the content of a cache line from a single DLOG entry. This assumption can be relaxed in the implementation without much engineering effort.

When detecting n_f fails, all non-failed nodes would remove n_f from all share lists where n_f appears such that n_f would not be a sharing node of any cache line. Thus, the data recovery only needs to process the data for which n_f is the home node or the owner node. We logically divide data recovery into two phases, which in practice can be performed in parallel. In the first phase, n_f recovers the data for which it is the home node, and in the second phase, each of those non-failed nodes recovers its dirty cache lines that are owned by n_f. During the recovery process, the locks that are currently held by non-failed nodes should be reserved in order for the applications running on those nodes to proceed. After data recovery, we shall show how to restore the directory for the cache lines of n_f that are currently locked by other nodes.

In the first phase of data recovery, n_f performs a reverse log scan, and makes use of each log entry that corresponds to an unrecovered cache line. For each such entry e, if it is logged by DLOG, n_f will directly use the content of e to restore the corresponding cache line c. Otherwise, n_f will ask the owner node of c to validate whether the counter of its most recent write of c is same as the value recorded in e. If yes, then n_f will promote that node as the owner of c, and accordingly update the cache directory. Otherwise, n_f logs an UNDO entry to invalidate e. This happens when, for example, in the scenario of Figure 4c, the home node n_h crashes after OLOG. In this case, failing to receive the acknowledgement, the request node n_r will never perform DLOG, leading to mismatched counters between n_r and n_f. It should be noted that each cache line will experience at most two recovery trials, since its second to last undone log entry is always valid.

The second phase is similar to the first one. For each dirty cache line owned by n_f, a non-failed node asks n_f to validate whether its most recent write of that cache line is consistent with the current counter. Depending on the validation result, this node will either promote n_f as the owner node or resort to the second to last undone log entry which it logged for that cache line.

After the data have been recovered, the cache lines of n_f that were previously "Shared" before the failure of n_f have been changed to "Dirty" or "UnShared", which means the shared locks on those cache lines acquired by other nodes have been implicitly released. As a result, we need to restore the directory for those cache lines. To that end, after data recovery, each non-failed node notifies n_f of the cache lines that n_f previously shared with it. For each such cache line, n_f will read the content from the current owner node (if possible), change its status to "Shared", and finally update its share list accordingly. We do not need to recover the exclusive locks, since as mentioned, each exclusive lock acquisition corresponds to

a DLOG entry, and hence the lock holder will always be promoted as the owner of the respective cache line.

The above recovery process can be applied to the case where multiple nodes fail simultaneously. In that case, the recovery process is deferred until all failed nodes become online and then performed for each of them in parallel. The process is analogous to the case of single node failure. Specifically, each failed node first recovers cache lines for which it is the home node from itself and non-failed nodes. This leaves unrecovered the cache lines whose home node and owner node are two different failed nodes. We defer their recovery to the second phase during which each failed node recovers its owned cache lines and hence restores the cache structure.

6. APPLICATIONS

In this section, we demonstrate how to develop applications using GAM via two examples: a transaction engine and a distributed hash table (DHT).

6.1 Transaction Engine

The shared memory model makes it easier to implement a distributed transaction engine by hiding the complicated network communication, and hence allows developers to focus only on the core single-node transaction processing logic. Basically, every transaction processing node has a root entry for the global index whose pointers are global addresses, so that it can access all tables by traversing the global index using the `Read` and `Write` APIs. For transaction processing protocol, we simply adopt the traditional two-phase locking (2PL) by using `TryRLock`/`TryWLock` APIs. We do not rely on two-phase commit (2PC) like protocol to achieve distributed consensus, since at the time a transaction is ready to commit, data is already acquired by the request node. By utilizing the `Lock` synchronization primitives, which issues `MFence` implicitly, we can achieve serializability easily under the PSO consistency model provided by GAM. Our transaction engine can avoid the 2PC overhead naturally, as a result of the unified memory model provided by GAM, and completely eliminates the complexity of data transmission in the application layer.

6.2 Distributed Hash Table

Based on the shared memory model provided by GAM, a DHT can be implemented as a distributed array of buckets wrapped across multiple nodes. Specifically, each GAM node is responsible for a subset of 64-bit key space, and the mapping between a key and its resident node is determined by the highest bits. In each bucket, there are multiple 12-byte entries and an overflow pointer to handle hash conflicts and enhance occupancy of the hash table. Each hash table entry contains a 12-bit tag extracted from the lowest 12 bits of the key to distinguish between keys in the same bucket, a 20-bit integer recording the size of the indexed key-value pair, and a 64-bit pointer pointing to the global address where the key-value pair is stored. In this way, unlike traditional DHT where hash tables and their indexed key-value pairs are collocated within the same physical node, our DHT implementation decouples key-value pairs from their indexing entries, and hence is able to not only reduce the cost of DHT updates (can re-allocate locally regardless of the original mapping), but also automatically balance the load of nodes in GAM.

7. PERFORMANCE EVALUATION

This section presents a performance study of GAM. We first introduce the settings for the experiments, and then conduct a micro benchmark and a macro benchmark to thoroughly profile GAM.

Table 3: Workload parameters

name	definition
read ratio	the percentage of `Read`/`RLock`
remote ratio	the percentage of remote accesses
spatial locality	the probability of an operation accessing a same cache line as the previous one
sharing ratio	the percentage of operations that access data shared across all nodes

The experiments are conducted on a cluster of 20 Supermicro 6018R-MiT 1U servers, each of which is equipped with a 3.5GHz quad-core Intel Xeon E5-1620 V3 CPU, 32 GB DDR3 memory and a 40 Gbps Mellanox MCX353A-QCBT InfiniBand adapter connected to a 40 Gbps Intel True Scale Fabric 12300 switch. These servers run an x86_64 Ubuntu 14.04 with a linux 3.13.0 kernel. The InfiniBand adapters are driven by Mellanox OFED 3.2-2.0.0.0. We also enabled the IPoIB Protocol which we rely on to run experiments for systems built atop traditional TCP/IP protocol stack.

7.1 Micro Benchmarks

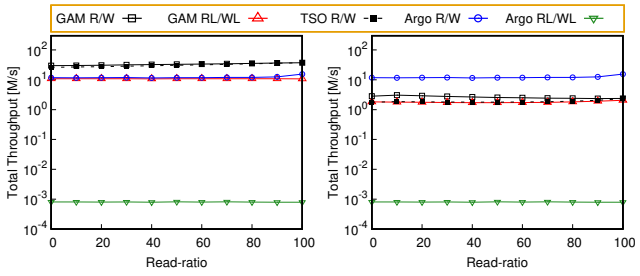
We deploy GAM on 8 nodes. Each node contributes its free memory to the global space, and employs a LRU cache which is configured to accommodate roughly half of its own working set (i.e., the objects accessed in the workload). For benchmark, each node launches a process to independently generate three types of workload: *R/W*, *RL/WL* and *RL+R/WL+W*. The *R/W* and *RL/WL* workloads respectively consist of `Read`/`Write` and `RLock`/`WLock` operations. The *RL+R/WL+W* workload is a combination of the other two workloads where each `Read`/`Write` is issued after acquiring the corresponding lock. Each operation in the workloads accesses an 8-byte object in the global memory. For the two lock workloads, each accessed object is unlocked at the end of the respective operation. The access pattern of the workloads is controlled via four parameters: *read ratio*, *remote ratio*, *data locality*, and *sharing ratio*, whose definitions are given in Table 3. Basically, the remote ratio and data locality jointly control how the accessed objects are distributed among the entire global memory space. The sharing ratio determines the confliction between 8 benchmark processes. By default, the objects accessed by each benchmark process are randomly distributed among the whole global space (i.e., remote ratio = 7/8 and data locality = 0), and do not overlap with those of other processes (i.e., sharing ratio = 0).

For comparison, we use two baselines, Argo [24] and GAM-TSO. GAM-TSO is a variant of GAM which appends a MFence operation to each write to disable write reorder and hence enforce TSO consistency model. We compare GAM against Argo [24] and GAM-TSO in terms of the throughput achieved under different workloads. Since GAM-TSO only affects the processing of write requests, its lock throughput is the same as GAM, and hence omitted. For each system, we run the benchmark four times for each workload parameter configuration. The first run is used to warm up the cache, and the result of the other runs is averaged for report.

7.1.1 Read Ratio

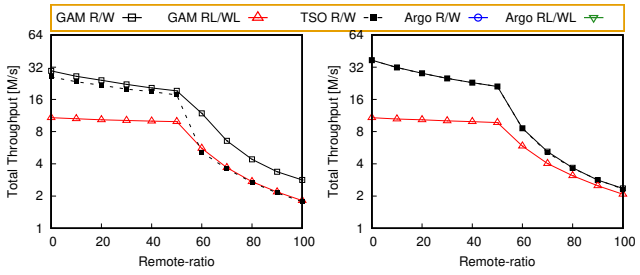
We first study how the read ratio affects the performance. As shown in Figure 6a, when there is no remote access, i.e., remote ratio is 0%, read ratio has no effect on the performance of GAM and GAM-TSO. The performance gap between the *R/W* workload and the lock workload of the two GAM systems is due to the additional overhead incurred by lock operations to enforce lock semantics.

When there are full of remote accesses (remote ratio = 100%), due to the limited cache size, some of the read/write requests cannot be absorbed by the cache layer of the two GAM systems, which



(a) Remote ratio = 0% **(b)** Remote ratio = 100%

Figure 6: Read Ratio



(a) Read ratio = 0% **(b)** Read ratio = 100%

Figure 7: Remote Ratio

hence leads to inter-node communications and significantly reduces the throughput of GAM and GAM-TSO. However, unlike the case of 0% remote ratio, GAM now performs much better (60%) than GAM-TSO when write requests are majority. This can be attributed to two features enabled by the PSO consistency model: 1) each GAM node can have multiple write requests being processed simultaneously, which improves the utilization of network resource and reduces CPU idle time, compared with TSO consistency, and 2) PSO allows a cache-hitting write request to be completed before an earlier cache-missing write request, which can improve cache hit ratio by effectively avoiding evicting cache lines that will be accessed in near future. As we observed, in the write workload, GAM achieves a cache hit ratio of 80%, which is 60% higher than that of GAM-TSO. When read requests start to dominate, the performance gap between GAM and GAM-TSO gradually shrinks due to the synchronous processing of read requests, and finally vanishes at 100% read ratio where the read/write performance of both systems converges to the lock performance.

For global memory allocation, Argo interleaves the physically allocated memory among all nodes, but provides no support for altering this behavior. Hence, the working set of each node is always evenly distributed among the cluster, However, since Argo does not allow to configure the cache size, each Argo node will cache all the working set after the first warming-up run, and is thus able to process each read/write request without incurring network communication in the following runs. Therefore, it makes little sense to compare the read/write throughput between Argo and GAM, and we thus omit the discussion of this comparison throughout the entire section. For the lock workload, Argo's performance is four orders worse than that of GAM and GAM-TSO. This evidences that Argo is only suitable for data-race-free applications, but will result in significant performance issues when applied to applications with medium-to-high data race, as will be shown in Section 7.3.

7.1.2 *Remote Ratio*

Figure 7 gives the throughput that GAM and GAM-TSO achieve under various remote ratios. Argo's result is not available since it is unable to adjust the remote ratio of the workloads for Argo. As shown in Figure 7, for both GAM and GAM-TSO, the performance of each workload only slightly degrades when the remote ratio increases from 0 to 50%, and then drop significantly thereafter. This is because as mentioned, the cache at each node can hold roughly half of its working set, and thus, when the remote ratio $\leq$ 50%, each node is able to cache all remote data in its working set after the warming-up run. Therefore, the initial slight performance degradation is due to increasing accesses of cache-related data structures, and the following sharp degradation is because of cache misses and their incurred inter-node communications.

For the write workload (Figure 7a), the cache miss incurred performance degradation of GAM is much smaller than that of GAM-TSO. As explained before, this is because the PSO consistency allows parallel, reordered processing of write requests. The write and the lock performances of GAM-TSO keep overlapping with each other since the remote ratio exceeds 50%. This is because both types of requests are processed sequentially in GAM-TSO, and incur additional overhead to respectively enforce memory fence and lock semantics. Without memory fences involved, read workloads are processed in exactly the same way for both GAM and GAM-TSO, and hence the same performance is achieved.

7.1.3 *Data Locality*

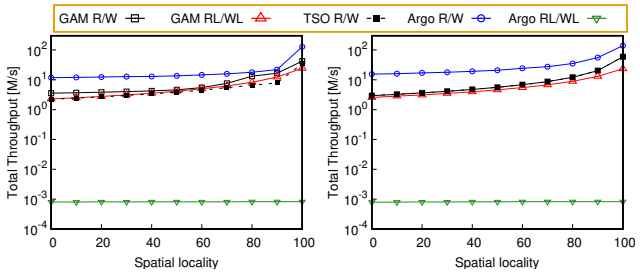
In this experiment, we study how data locality, which is the key performance booster for systems with caching, affects the performance of GAM and the two baseline systems. The result is shown in Figure 8, where the effect of temporal locality is omitted since it is similar to that of spatial locality.

With higher data locality, both of the read/write and the lock performance of the two GAM systems can be substantially improved due to higher global cache hit ratio. Argo's read/write performance also benefits from higher locality. Unlike GAM, Argo's performance improvement results from higher utilization of CPU cache, as Argo can always cache the entire working set after the initial warming-up run, and hence achieve a 100% global cache hit ratio. The lock performance of Argo is bottlenecked by costly synchronizations, and is thus insensitive to data locality.

The performance gap between the write and the lock workloads of GAM exhibits an interesting trend with respect to data locality: it gradually diminishes until vanishing at 60% data locality, and starts to enlarge since then. This is because, as aforementioned, for the write workload, the PSO consistency enables GAM to achieve a high cache hit ratio even in low-locality cases. Hence, the write workload cannot benefit as much as the lock workload from improved data locality, which explains the initial diminishing of their performance gap. From 60% onwards, GAM achieves the same cache hit ratio for both the write and the lock workloads. However, since the write workload can exploit CPU cache more effectively than the lock workload, it can benefit more from further increase of data locality. For GAM-TSO, its write performance is similar to the lock performance, which can also be observed in the remote ratio experiment, and attributed to the same reasons mentioned therein.

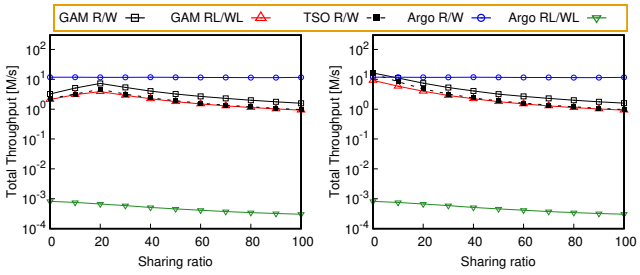
7.1.4 *Sharing Ratio*

Sharing ratio is one of the most sensitive factors to the performance of GAM, as data accesses (especially write-related operations) to the shared data may cause frequent inter-node communications, which will offset the performance gain drawn from the distributed cache. In this experiment, we set the read ratio to 50% to stress the GAM systems since there will be a lot of invalidations generated.



(a) Read ratio = 0% **(b)** Read ratio = 100%

Figure 8: Data Locality



(a) Cache size = 50% of workload **(b)** Cache size = 100% of workload

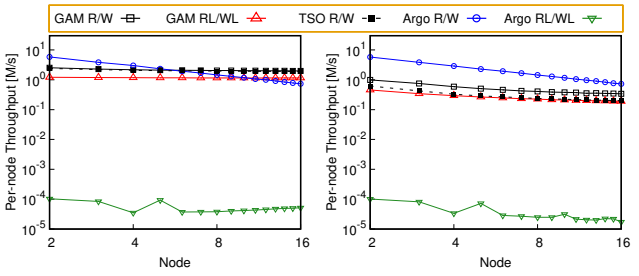
Figure 9: Sharing Ratio

For Argo, its read/write performance is not influenced by the sharing ratio, since Argo is able to cache the entire working set locally and incurs no synchronization while performing the *R/W* workload. Due to added cache invalidations, one may expect the performance of the two GAM systems to constantly deteriorate with higher sharing ratio. Interestingly, this is not the case. As shown in Figure 9a, the performance of the two systems experiences an obvious improvement during the initial increase of the sharing ratio. This is because of improved cache exploitation. Specifically, in the cases of small sharing ratios, the shared cache lines are less frequently accessed than other cache lines, and thus are better candidates for eviction. As a result, when those shared cache lines are invalidated by conflicting write operations, the cache efficiency is essentially improved!

In order to verify the above claim, we increase the cache size of GAM nodes such that the cache can hold the entire working set. As shown in Figure 9b, for the two GAM systems, the throughput now, as expected, constantly decreases with higher sharing ratio, due to more incurred invalidations. By comparing Figure 9a and 9b, we can also estimate how cache size affects GAM's performance. When there is no sharing access, the performance of the GAM systems is very sensitive to cache size: compared to the default case (Figure 9a), doubling the cache size can improve the performance of GAM and GAM-TSO by 4x and 6x, respectively. However, when the sharing ratio is high (e.g., $\geq 40\%$), the performance of the GAM systems is bottlenecked by the invalidations, and cannot be improved by larger cache space.

7.1.5 Scalability

For this experiment, we investigate how the performance of each system varies with the number of nodes involved in global memory. As in previous experiments, the working set of each node is uniformly distributed among the entire global memory. Consequently, in the default setting of cache size, the remote ratio, $(N-1)/N$, varies with the number of nodes N. In order to exclude the effect of the changing remote ratio, we configure the cache size such that each node is able to cache its entire working set locally. The read ratio is set to 50%, and two sharing ratio configurations are chosen: 0% for invalidation-free case and 50% for invalidation-heavy case.



(a) Sharing ratio = 0% **(b)** Sharing ratio = 50%

Figure 10: Scalability

Figure 10 presents the performance of different systems under the above workload configurations. As shown in this figure, the two GAM systems can scale almost linearly when there is no sharing access, which is obvious since no invalidation is generated, and each node can have all its working set locally cached after warming-up. Strangely, Argo does not scale well in this case, although each node can also cache the entire working set and incur little overhead for cache coherence. In the case of 50% sharing ratio, the performance of the GAM systems does get impaired due to frequent read/write confliction and the incurred invalidation. On the other hand, the degree of performance degradation of the GAM systems is slightly alleviated as more and more nodes are added. This is because a home node, upon receiving a remote write request, can send the invalidations to all the sharing nodes simultaneously, which improves network utilization.

7.2 Distributed Transaction Processing

In this experiment, we use TPC-C benchmark suite [43] to compare the transaction engine of GAM against L-Store [27], FaRM [14] and Tell [30]. L-Store tries to transfer all the data required by a transaction to the request node to avoid the two-phase commit (2PC), which has shown better performance than H-Store [22]; FaRM also provides the global memory, but only supports transactional memory accesses that are based on 2PC; Tell uses an RDMA-aware storage system, RAMCloud [36]), as the underlying distributed store, and is hence able to natively exploit fast RDMA networking.

For this experiment, all data, including tables and indices, are uniformly distributed in the global memory space of GAM, and the transaction engine is UNAWARE of the underlying data distribution. But in order to study the *distribution ratio* in the experiment, the transaction engine enables all the clients in the same warehouse to issue transactions to the same server node and access the data in other server nodes with a probability controlled by the *distribution ratio*. It should be noted that we do not use this extra application knowledge to optimize the transaction processing of GAM, such as co-partitioning data and index, or replicating read-only data[4].

We only run the new-order and payment transaction procedures of the TPC-C benchmark suite, as all the other three transaction procedures are local. We deploy all the systems on 8 nodes, with 4 threads in each node. For Tell, the cluster setups are: 5 processor nodes, 2 storage nodes and 1 node for the commit manager. For FaRM, we implement their protocol in our GAM code base,

[4]Unless otherwise specified, we also do not employ these optimizations in the transaction layer for other systems.

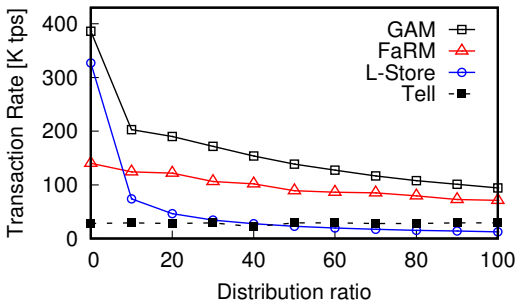


Figure 11: TPC-C Benchmark

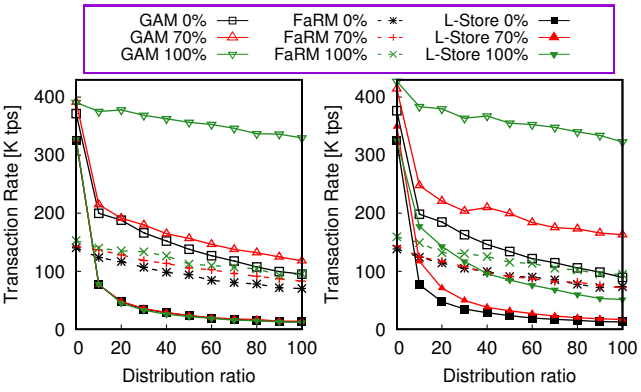
as their source code is not available[5]; for L-store and Tell, we run their open-sourced codes to obtain the performance results. As we can see, GAM performs the best under different *distribution ratios*, but its performance converges to that of FaRM when the *distribution ratio* is high (e.g., more than 70%). Actually, we find that the default TPC-C workload represents the worst case for GAM, since TPC-C benchmark is write-intensive, and most of the working set will be shared due to its non-uniform access pattern, which, as shown in Section 7.1.4, would lead to a lot of cache invalidation, especially in cases with a high *distribution ratio*.

FaRM does not perform well when *distribution ratio* is low, which is mainly due to the fact that there is still some data (e.g., ITEM table) that cannot be co-partitioned with other tables (e.g., STOCK table) , even though we have already manually co-partitioned the index for FaRM[6]. L-Store performs well when there are few distributed transactions, but its performance drops significantly even when there are only a small amount of distributed transactions, which is mainly because of the network overhead in L-Store. Specifically, L-Store uses traditional TCP/IP sockets for message delivery, and in order to make it work with our InfiniBand network adapters, we run L-Store over the IPoIB protocol, which, however, incurs a lot of protocol overhead. We do not modify L-Store to use native RDMA *verbs* due to the completely different APIs between TCP/IP sockets and RDMA *verbs*. The performance of Tell does not change with *distribution ratios*, since its storage is decoupled with the transaction engine, leading to each data access to be fulfilled via RDMA networking.

We further analyze the performance on TPC-C with the varied read ratio and temporal locality. The read ratio is set as the probability to change a write on a record to a read in the running procedures; the temporal locality is set as the probability to access a data item in the last transaction when generating the access item set for the current transaction. As shown in Figure 12a, when we increase the *read ratio*, the performance of GAM is improved gradually. When all transactions are read-only, the performance remains almost unchanged with increased *distribution ratios*. In contrast, larger *read ratio* leads to a much less improvement in the performance of FaRM than that of GAM, and does not change the performance of L-Store at all since L-Store has the same strategy for both read and write operations. A similar trend is also observed for *temporal locality*, which is more beneficial to GAM than others, as shown in Figure 12b. Even though L-Store adopts similar "caching" idea as GAM and eliminates the 2PC overhead, its performance cannot be improved significantly with increasing *temporal locality* in the presence of distributed transactions, as a result

[5]The performance results are consistent with the published results [15] with a larger cluster of higher-end servers.

[6]This is the extra bonus to FaRM, as we do not optimize for GAM.



(a) Read Ratio **(b)** Temporal Locality

Figure 12: TPC-C Benchmark – Analysis

of its inefficient network communication[7]. We do not show the result for Tell, as its performance does not change with these factors. In conclusion, with these cache-friendly characteristics (i.e., high read ratio and temporal Locality) in the workload, GAM performs significantly better than others.

7.3 Distributed Hash Table

We compare our GAM-based DHT implementation with the DHT implementations built on top of other RDMA-based shared memory frameworks, including Grappa [35] and Argo [24]. We use these DHTs to run the YCSB benchmark [12] over 8 nodes. The Zipfian distribution parameter of the YCSB benchmark is set to the default value, 0.99. The results with varied *get ratios* and *number of threads*[8] are shown in Figure 13.

It is obvious that GAM-based DHT is superior to Grappa-based DHT in all scenarios. For update-heavy workloads, GAM-based DHT performs 0.4-2.2x better than Grappa-based DHT. With increasing get requests in the workload, thanks to the existence of cache in GAM and its elimination of remote accesses, the throughput of GAM-based DHT is significantly improved. In contrast, since Grappa does not employ any cache mechanism, the major cost for both types of requests (i.e., get and put), is spent in remote accesses, leading to the same throughput for both the update-heavy and read-heavy workloads. Consequently, the performance gap between the two DHTs enlarges substantially with the increase of the *get ratio*, and approaches 22x in the scenario with four threads running pure get workloads.

As shown in the figure, Argo-based DHT performs not so well in all scenarios. This is because Argo optimizes for data-race-free applications and exploits coarse-grained synchronization primitives in the critical sections which incur a huge overhead at synchronization points. Specifically, when acquiring or releasing a lock, Argo will invoke a heavy-weight `acquire`/`release` procedure that updates the status of all the cached data. Since each node of Argo maintains a read list and a write list for each cache line for which it is the home node, a request node, upon `acquire`, will instruct each home node to remove itself from all respective read lists. For `release`, as we have mentioned, each node needs to write back the dirty cache lines to the home nodes. Since DHT is a

[7]Even at 100% *temporal locality*, there are still some remote data accesses as the *temporal locality* is based on the last transaction, if the current transaction has a larger item set than the last one, it can introduce new data items.

[8]Since Grappa is single-threaded, we increase the number of instances per node using MPI rather than the number of threads inside one instance.

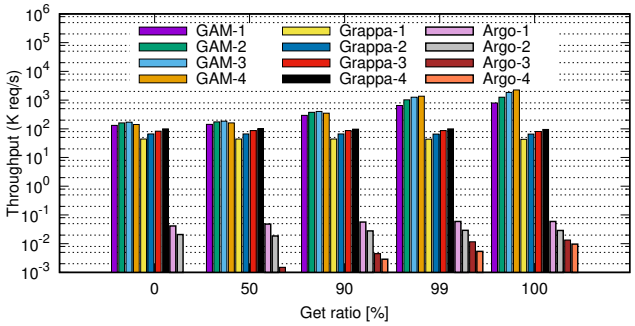


Figure 13: Single-node DHT performance with varying number of threads

data-race-frequent application which requires locking for each key lookup, the high overhead incurred by the synchronization significantly impairs the performance of Argo. Compared with Argo, GAM is efficient to implement such applications because of its fine-grained synchronization primitives which support locking operations on object level, and are more light-weight by avoiding the node-wide synchronization in Argo.

8. RELATED WORK

Providing a unified memory model abstracted from physically distributed memory has long been researched. Due to slow network access, traditional systems [4, 7, 26, 40] typically employed release consistency [1] to amortize network latency with larger message size at synchronization points by requiring dirty data to be made visible only at the next release. An efficient implementation of release consistency is given in [26] for loosely coupled networked clusters. TreadMarks [4] presented a lazy release consistency model to reduce the cost of synchronization by flushing dirty cache in an on-demand manner, while Cashmere-2L [40] relied on a customized network to achieve the same purpose. Munin [7] allowed more flexibility on the enforcement of the consistency model at a cost of manual annotations. The release consistency employed in these systems renders them more appropriate for data race free applications due to the high synchronization cost, while GAM, by exploiting RDMA to enforce cache coherence, is suitable for a much wider range of applications. Some other works [16, 17] organize the distributed memory as a cache layer of local disk to reduce slow disk accesses, which is different from the global memory model as provided in GAM. In addition, there were several works [3, 8, 9, 11, 49] providing the global memory abstraction at the language level, which however imposes a burden of learning a new language on users.

There are recent interests in distributed memory management with RDMA [39, 24, 14, 2, 35, 30, 5]. DSPM [39] is an RDMA-aware distributed shared memory which leverages non-volatile memory (NVM) to provide data persistence. Argo [24] designed a set of relaxed cache coherence protocols based on RDMA-aware MPI [29] to reduce the synchronization overhead of enforcing release consistency in global memory space. Both of DSPM [39] and Argo [24] still require users to manually call synchronization primitives for data consistency, which can incur a significant overhead in the presence of data contention, as we have shown in Section 7.3. Sinfonia [2] and FaRM [14] used two-phase commit protocol to provide transaction support for global memory operations. GAM does not provide such a support, but provides a set of synchronization primitives to allow users to build transactional applications. As we show in 7.2, transaction engines on top of GAM can benefit from the cache coherence protocol and convert the distributed transactions into a local transaction, thus exhibiting better performance than 2PC based protocols. Grappa [35] was designed for a latency-tolerance tasking framework and a shared memory abstraction for irregular data-intensive applications with poor locality such as graph analytics, and hence did not employ any caching mechanism. In contrast, GAM is designed to be applicable to a variety of applications with performance guarantee by exploiting data locality. Tell [30] and NAM [5] also proposed RDMA-based shared-memory architectures. They decoupled query processing and data storage into two separate layers, so that (1) each layer can scale out independently and (2) there is no need to handle load imbalance of data. However, like many other shared-nothing databases [22, 42, 41], such design is unable to exploit the data locality. GAM addresses this limitation via an RDMA-optimized cache coherence protocol, which can help to improve the performance for various workloads as shown in Section 7.

RDMA has been used to improve the performance of various distributed systems, such as distributed file systems [20, 47], by enabling RDMA in the communication substrate. There have also been many works on using RDMA to boost the performance of database systems [25, 18, 46, 45]. The first two works [25, 18] used RDMA to combine distributed memory together to augment memory budget for database systems. DrTM [46] combines RDMA and HTM (Hardware Transactional Memory) to boost the performance of distributed transaction processing. An RDMA-aware data shuffling operator was designed in [28] for parallel database systems. Query Fresh [45] used RDMA to accelerate the shipment of log records between the master and backup servers. In addition, RDMA was also heavily exploited for key/value store [21, 33, 36]. RAMCloud [36] followed the classic design such that two-sided `Send` verbs are used for request processing, which involves both the client and the server. Pilaf [33] relieved the server from being involved in processing of `get` requests by allowing clients to read key/value pairs using one-sided RDMA `Read` verbs, and designed a self-verifying data structure to avoid read-write race condition. HERD [21] adopted a different design such that the server handles all requests by polling pre-determined memory address where incoming request will be written by clients via one-sided RDMA `Write` verbs. These systems exploited RDMA for a single application, whereas GAM is a generic distributed memory platform which allows various applications to be built atop.

9. CONCLUSIONS

In this paper, we propose GAM, a distributed memory management platform which provides a unified memory model abstracted from the distributed memory interconnected with RDMA network. Enabled by RDMA, an efficient distributed cache coherence protocol is designed and employed in GAM to exploit the locality in global memory access. Based on the PSO consistency model, `Write` operations are made asynchronous and pipelined, hiding most of the latency incurred in the complicated write protocol. A logging scheme is designed for GAM to survive node failures without incurring much overhead. The transaction engine and DHT built on top of GAM have demonstrated its strengths in terms of programming simplicity, applicability and performance. With increasingly faster networking, distributed memory sharing may hold the key to future distributed computing.

Acknowledgements

This research was supported by the National Research Foundation, Prime Ministers Office, Singapore, under its Competitive Research Programme (CRP Award No. NRF CRP8-2011-08). Gang Chen's work was supported by the National Basic Research Program (973 Program, No. 2015CB352400).

10. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, et al. The fortress language specification. *Sun Microsystems*, 139(140):116, 2005.

[4] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, Feb 1996.

[5] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[6] Q. Cai, H. Zhang, W. Guo, G. Chen, B. C. Ooi, K. L. Tan, and W. F. Wong. Memepic: Towards a unified in-memory big data management system. *IEEE Transactions on Big Data*, pages 1–1, 2018.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *SOSP '91*, pages 152–164, 1991.

[8] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[10] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47. ACM, 2005.

[11] U. Consortium et al. Upc language specifications v1. 2. *Lawrence Berkeley National Laboratory*, 2005.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC '10*, pages 143–154, 2010.

[13] P. J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, 2005.

[14] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *NSDI '14*, pages 401–414, 2014.

[15] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *SOSP '15*, pages 54–70, 2015.

[16] M. J. Feeley, W. E. Morgan, E. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 201–212. ACM, 1995.

[17] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server database architectures. In *VLDB*, pages 596–609, 1992.

[18] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, pages 649–667, 2017.

[19] InfiniBand Trade Association. Infiniband roadmap. http://www.infinibandta.org, 2016.

[20] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 35. IEEE Computer Society Press, 2012.

[21] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM '14*, pages 295–306, 2014.

[22] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[23] A. K. M. Kaminsky and D. G. Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference*, page 437, 2016.

[24] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros, and K. Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 3–14. ACM, 2015.

[25] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and rdma. In *SIGMOD '16*, pages 355–370, 2016.

[26] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *TOCS*, 7(4):321–359, Nov. 1989.

[27] Q. Lin, P. Chang, G. Chen, B. C. Ooi, K.-L. Tan, and Z. Wang. Towards a non-2PC transaction management in distributed database systems. In *SIGMOD '16*, pages 1659–1674, 2016.

[28] F. Liu, L. Yin, and S. Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. In *EuroSys*.

[29] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High performance RDMA-based mpi implementation over infiniband. In *ICS '03*, pages 295–304, 2003.

[30] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the design and scalability of distributed shared-data databases. In *SIGMOD '15*, pages 663–676, 2015.

[31] Mellanox. Connectx©-6 en 200gb/s adapter. http://www.mellanox.com/related-docs/prod_silicon/PB_ConnectX-6_EN_IC.pdf, 2016.

[32] Mellanox. Infiniband performance. http://www.mellanox.com/page/performance_infini-band, 2016.

[33] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX ATC '13*, pages 103–114, 2013.

[34] B. Mutnury, F. Paglia, J. Mobley, G. K. Singh, and R. Bellomio. Quickpath interconnect (QPI) design and

analysis in high speed servers. In *EPEPS '10*, pages 265–268, 2010.

[35] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC '15*, pages 291–305, 2015.

[36] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *Operating Systems Review*, pages 92–105, 2010.

[37] QLogic. Introduction to Ethernet latency. http://www.qlogic.com/Resources/Documents/TechnologyBriefs/Adapters/Tech_Brief_Introduction_to_Ethernet_Latency.pdf, 2016.

[38] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *PVLDB*, 9(4):228–239, 2015.

[39] Y. Shan, S.-Y. Tsai, and Y. Zhang. Distributed shared persistent memory. In *SoCC*, pages 323–337, 2017.

[40] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software coherent shared memory on a clustered remote-write network. In *SOSP '97*, pages 170–183, 1997.

[41] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[42] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *IEEE Data Engineering Bulletin*, 2013.

[43] Transaction Processing Performance Council. TPC-C benchmark specification. http://www.tpc.org/tpcc, 2010.

[44] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. *PVLDB*, 11(10):1137–1150, 2018.

[45] T. Wang, R. Johnson, and I. Pandis. Query fresh: Log shipping on steroids. *PVLDB*, 11(4):406–419, 2017.

[46] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104. ACM, 2015.

[47] J. Wu, P. Wyckoff, and D. Panda. Pvfs over infiniband: Design and performance evaluation. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 125–132. IEEE, 2003.

[48] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32. ACM, 2007.

[49] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, et al. Titanium: A high-performance java dialect. *Concurrency Practice and Experience*, 10(11-13):825–836, 1998.