Designing Far Memory Data Structures: Think Outside the Box

Marcos K. Aguilera
VMware

Kimberly Keeton
Hewlett Packard Labs

Stanko Novakovic
Microsoft Research

Sharad Singhal
Hewlett Packard Labs

Abstract

Technologies like RDMA and Gen-Z, which give access to memory outside the box, are gaining in popularity. These technologies provide the abstraction of *far memory*, where memory is attached to the network and can be accessed by remote processors without mediation by a local processor. Unfortunately, far memory is hard to use because existing data structures are mismatched to it. We argue that we need new data structures for far memory, borrowing techniques from concurrent data structures and distributed systems. We examine the requirements of these data structures and show how to realize them using simple hardware extensions.

ACM Reference Format:
Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing Far Memory Data Structures: Think Outside the Box. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321433

1 Introduction

Far memory technologies, such as RDMA [33] and Gen-Z [15], which allow memory to be accessed via an interconnect in a cluster, are generating much interest. The fundamental capability of far memory is *one-sided access*, with which processors in the cluster can directly read and write far memory without mediation by a processor close to the memory. Far memory brings many potential benefits over near memory: higher memory capacity through disaggregation, separate scaling between processing and far memory, better availability due to separate fault domains for far memory, and better shareability among processors.

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321433

However, far memory has higher latency than near memory, which poses challenges. Developers often use memory through high-level data structures, such as hash tables, trees, and queues. Unfortunately, existing data structures are ill-suited for use with far memory because they were designed under different assumptions. *Data structures for local memory* assume a small, homogeneous access time to memory, which does not hold for far memory. *NUMA-aware data structures* discriminate between local and remote socket memory, but they assume cache coherence and strive to minimize coherence traffic across sockets. Far memory, however, does not guarantee cache coherence. *Distributed data structures* assume that memory is operated on by a local processor, where remote processors access that memory via RPCs and similar mechanisms. Far memory is fundamentally different, working under one-sided accesses. So, there is a mismatch between today's data structures and the ones needed for far memory.

Prior work [24, 25, 35] demonstrates this mismatch, by considering traditional hash tables that are implemented with two approaches: (1) using far memory with one-sided access, or (2) using traditional memory with two-sided RPC access. The authors show that (2) outperforms (1), concluding that one-sided access appears to have diminished value. We interpret these results differently: traditional hash tables, as is, are the wrong data structure for far memory.

We need new data structures designed specifically for far memory that consider its assumptions and performance. In particular, data structures need operations that complete in $O(1)$ far memory accesses most of the time, preferably with a constant of 1 (§3.1). This requirement precludes most existing data structures today from use with far memory. For instance, linked lists take $O(n)$ far accesses, while balanced trees and skip lists take $O(\log n)$.

To reduce far accesses, we must trade them for near accesses, by caching some data intelligently at clients, by optimizing data structure traversals across far memory, and by efficiently supporting data sharing in the face of caching.

Furthermore, we believe that far memory needs broader hardware primitives to properly support efficient data structures. Inspired by ideas from processor instruction set architectures (ISAs) and from distributed systems, we propose simple hardware extensions to far memory that are powerful and generally applicable, in that they avoid round trips to far memory, they permit a consistent view of data across nodes, and they

work on many data structures. Specifically, we propose that far memory support *indirect addressing*, general *scatter-gather* primitives, and *notifications*. Indirect addressing is a common feature of processors; it permits a single instruction to deference a pointer and load or store at the resulting address; more sophisticated versions have an index that specifies an offset added to a base address to choose the pointer. Scatter-gather combines many memory transfers into one, without requiring explicit management by application or system software. Notification permits a user to know that a location has changed without continuously reading that location. These primitives are useful for far memory because they avoid far memory round trips, while providing the required functionality to improve data structure caching and traversal.

We identify how these extensions support new data structures for far memory, such as counters, vectors, maps, and queues (§5). We illustrate the benefits of far memory data structures with a simple case study of a monitoring application (§6). We examine in detail its data structure needs and show how they map to the primitives that we identify. We conclude the paper with a discussion of how to deal with issues of scale in implementing the proposed hardware primitives (§7).

2 Background

Recent advances provide the building blocks for constructing architectures with a large pool of *far* memory that can be shared across a high-performance system interconnect by decentralized compute resources. High-performance system interconnects, such as RDMA [33], OmniPath [21], and Gen-Z [15], provide low-latency access from processors to far memory. Existing systems can transfer 1 KB in 1μs using RDMA over InfiniBand FDR 4x [5, 30].

Commercial [14, 18, 19, 22, 23] and research proposals [1, 2, 5, 26, 28, 29, 32] for far memory architectures share several characteristics. First, they provide a high-capacity shared pool of far memory that can be accessed by processors at latencies within 10× of node-local *near* memory latencies. This far memory pool could be implemented using any of several technologies (e.g., an RDMA-enabled memory server or Gen-Z fabric-attached memory). Second, they provide a *partially disaggregated architecture*, in that they treat near memory as private and far memory as shared. We use near memory as a data structure-specific (i.e., non-system-managed) cache for shared data in far memory. Third, far memory has no explicit owner among application processors. Thus, access to far memory by one processor does not need to be mediated by any other processor. This unmediated access is provided by one-sided loads/stores or gets/puts and facilitated through atomic operations (e.g., compare-and-swap). Fourth, partial disaggregation provides separate fault domains between application processing and far memory, meaning that failure of a processor does not render far memory unavailable.

We make several assumptions about the capabilities of the far memory technologies. First, we assume the existence of atomic operations on far memory (e.g., compare-and-swap (CAS) as in RDMA [34] or Gen-Z [16]), which permit a processor to make an update to far memory in an all-or-nothing fashion. These operations provide atomicity at the fabric level, bypassing the processor caches and near memory. Second, we assume that the memory fabric can enforce ordering constraints on far memory operations. In particular, memory operations issued before a memory barrier (or fence) complete before memory operations after the barrier. This functionality can be provided using request completion queues in the memory fabric interface.

3 What is a far memory data structure?

A far memory data structure has three components: (1) *far data* in far memory, containing the core content of the data structure; (2) *data caches* at clients, containing data that is discarded when clients terminate execution; and (3) an *algorithm for operations*, which clients execute to access the data structure. Here, *client* designates the user of the data structure.

Far memory data structures share some characteristics with both near memory and distributed data structures. Like near memory data structures, their operations are constrained to modify data using only hardware primitives. In contrast, distributed data structures can use mechanisms such as RPCs that can access many data items in arbitrary ways, with implications for both performance and concurrency. Like distributed data structures, a far memory data structure defines how the local memories should be accessed and changed to cache data. In contrast, near memory data structures do not directly manage their caches.

These differences suggest different performance metrics and requirements for far memory data structures, which we consider next.

3.1 Metrics for performance

Operations on far memory data structures incur both local and far accesses. Far accesses dominate the overall cost, as they are an order of magnitude slower (O(1μs)) than local accesses (O(100ns)). Moreover, local accesses can be hidden by processor caches, while far accesses cannot, which further increases the performance difference. For these reasons, the key performance metric for far memory data structures is far memory accesses.

To be competitive with distributed data structures using RPCs, far memory data structures should incur few far accesses, preferably one, most of the time. With too many far accesses, far memory data structures lose to distributed data structures in performance. With distributed data structures, a processor close to the memory can receive and service RPC requests to access the data structure. Doing so consumes the local processor, but takes only one round trip over the fabric. Far memory data structures that take several round trips to far memory perform worse. Fundamentally, this is the age-old trade-off between shipping computation or data, where

an RPC with traditional memory ships computation, while a one-sided access model with far memory ships data.

3.2 Requirements

We want general-purpose far memory data structures that can be broadly used across many applications. Each data structure has its own requirements, which vary along two dimensions: concurrent access model and freshness.

Concurrent access model: whether one or many clients can write to the data structure, and whether one or many clients can read it. The access model affects how clients manage their caches, with a more restrictive model improving efficiency. Additionally, concurrency requires techniques to avoid introducing inconsistencies in data.

Freshness: whether the data structure must return the most recent values or whether it can return stale data. This dimension affects how clients can use their caches.

4 Hardware primitives

While it is possible to design new far memory data structures using unmodified hardware, current hardware poses unnecessary obstacles. To achieve few far accesses per operation, we need to go beyond the current far memory primitives, which are limited to loads, stores, and a few atomics (§2). We need primitives that do more work, but not so much that they need an application processor to execute. In this section, we discuss such primitives and give an intuition of how they help; in the next sections (§5, §6), we use them to develop data structures.

We consider only hardware extensions with three characteristics: (1) they are relatively simple and have a narrow interface; (2) they make a significant difference; and (3) they are general-purpose. We consider three classes of extensions, leveraged from other domains: indirect addressing, scatter-gather, and notifications. We discuss scalability issues for implementing these hardware primitives in §7.

4.1 Indirect addressing

Common in processors, indirect addressing dereferences a pointer to determine another address to load or store. More precisely, given address *ad*, the hardware reads *ad* to obtain address ad', and loads or stores on ad' (*load0* or *store0* in Fig.1). Two useful variants add an index *i* to either *ad* or ad', which extracts a chosen field of a struct (*load1*, *store1*, *load2*, *store2* in Fig.1). Indirect addressing is useful for far memory, because it avoids a round trip when a data structure needs to follow a pointer.

Fetch-and-add is provided by some far memories, such as RDMA and Gen-Z. This instruction adds a value v to a location and returns its old value. Fetch-and-add-indirect adds v to a location and returns the value pointed by its old value (*faai* in Fig.1). Store-and-add-indirect stores a value v' instead of returning one (*saai*). These instructions provide the common idiom of dereferencing a pointer and incrementing it ("*ptr++").

Instruction	Definition
load0(*ad*, ℓ)	*tmp*= $*$ *ad*; return $*$*tmp*;
store0(*ad*, *v*, ℓ)	*tmp*= $*$ *ad*; $*$*tmp*=*v*;
load1(*ad*, *i*, ℓ)	*tmp*= $*$ (*ad* + *i*); return $*$*tmp*;
store1(*ad*, *i*, *v*, ℓ)	*tmp*= $*$ (*ad* + *i*); $*$*tmp*=*v*;
load2(*ad*, *i*, ℓ)	*tmp*=($*$*ad*) + *i*; return $*$*tmp*;
store2(*ad*, *i*, *v*, ℓ)	*tmp*=($*$*ad*) + *i*; $*$*tmp*=*v*;
faai(*ad*, *v*, ℓ)	*tmp*= $*$ *ad*; *ad* += *v*; return *tmp*;
saai(*ad*, *v*, v', ℓ)	$*$*ad* = v'; *ad* += *v*;
add0(*ad*, *v*, ℓ)	$**$ *ad* += *v*;
add1(*ad*, *v*, *i*, ℓ)	*tmp* = *ad* + *i*; $**$ *tmp* += *v*;
add2(*ad*, *v*, *i*, ℓ)	*tmp* = $*$*ad* + *i*; $*$*tmp* += *v*;
rscatter(*ad*, ℓ, *iovec*)	Read far memory range, store in local *iovec*
rgather(*iovec*, *ad*, ℓ)	Read far memory *iovec*, store in local range
wscatter(*ad*, ℓ, *iovec*)	Write far memory *iovec* from local range
wgather(*iovec*, *ad*, ℓ)	Write far memory range from local *iovec*
notify0(*ad*, ℓ)	signal change in [*ad*, *ad* + ℓ)
notifye(*ad*, *v*, ℓ)	signal *ad* set to *v*
notify0d(*ad*, ℓ)	signal change in [*ad*, *ad* + ℓ), return data

Figure 1. Primitives from other domains useful to far memory. All primitives include a length ℓ to determine how many bytes to transfer (not shown in pseudo-code). *iovec* is an array of buffers with a base pointer and length for each.

Additional useful variants add v to a value pointed to by a location (*add0*), possibly with indexing (*add1* and *add2*).

4.2 Scatter-gather

Scatter and gather permit clients to operate on disjoint buffers in one operation, without requiring explicit management in application or system software. With far memory, this primitive has four variants, depending on (a) whether we read or write, and (b) whether the disjoint buffers are at the client or in far memory (*rscatter*, *rgather*, *wscatter*, *wgather* in Fig.1).

Scatter-gather can be implemented on the client side. In fact, some operations are already provided in RDMA (*wgather*) or Gen-Z (*rgather* and *wscatter*). To gather from far memory or scatter to far memory, the client-side network adapter issues concurrent operations to far memory. To gather from local memory or scatter to local memory, the adapter issues a single operation to far memory using many local buffers.

4.3 Notifications

A notification is a callback triggered when data changes—an idea often used in cache coherence protocols and distributed systems when a remote party needs to monitor changes without having to constantly probe remotely. Far memory has that flavor, with a high cost to probe. Notifications can be used to invalidate or update cached data or metadata at clients, when the client wants to avoid stale content in its cache.

With far memory, a location refers to a memory address or a range of addresses (*notify0* in Fig.1). We also propose equality notifications as another useful extension. This notification is triggered when a word matches a given value (*notifye* in Fig. 1). Another variant returns the changed data (*notify0d*), which is useful when data is small.

For ease of implementation, *ad* and ℓ need to be word-aligned and not cross page boundaries, so that the hardware can associate notifications with pages (e.g., record them in page table entries at the memory node to avoid additional metadata). These are not strong limitations since notification clients (data structure algorithms) can align data as needed.

Because we want notifications to be scalable, they may be delivered in a best-effort fashion (e.g., with delay or unreliably). We discuss these scalability issues in §7.

5 Far memory data structures

In this section, we identify how our proposed hardware extensions support new data structures for far memory, including simple data structures (e.g., counters, vectors, mutexes and barriers), maps, queues, and refreshable vectors.

5.1 Simple data structures

Counters are implemented using loads, stores, and atomics with immediate addressing. *Vectors* take advantage of indirect addressing (e.g., *load1* and *store1*) for indexing into the vector using a base pointer. If desired, client caches can be updated using notifications: clients subscribe to specific (ranges of) addresses to receive notifications when they are modified (*notify0*) or when they reach arbitrary values (*notifye*).

Far memory *mutexes* and *barriers* are implemented similarly to their shared memory analogues. Mutexes use a far memory location initialized to 0. Clients acquire the mutex using a compare-and-swap (CAS). If the CAS fails, equality notifications against 0 (*notifye*) indicate when the mutex is free. Barriers use a far memory decreasing counter initialized to the number of participants. As each participant reaches the barrier, it uses an atomic decrement operation to update the barrier value. Equality notifications against 0 (*notifye*) indicate when all participants complete the barrier.

5.2 Maps

With near memory, maps are implemented using hash tables or trees. However, these are poor choices for large maps in far memory. Hash tables have collision problems: chaining within buckets leads to additional round trips, and inlining colliding items on a bucket limits the chain length but affects throughput. Furthermore, resizing hash tables is disruptive when they are large (e.g., a trillion elements). With trees, traversals take $O(\log n)$ far accesses; this cost can be avoided by caching most levels of the tree at the client, but that requires a large cache with $O(n)$ items (e.g., a B-tree with a trillion elements must cache billions of elements to enable single round trip lookups).

To address this problem, we propose a new data structure, the *HT-tree*, which is a tree where each leaf node stores base pointers of hash tables. Clients cache the entire tree, but not the hash tables. To find a key, a client traverses the tree in its cache to obtain a hash table base pointer, applies the hash function to calculate the bucket number, and then finally accesses the bucket in far memory, using indirect addressing to follow the pointer in the bucket. When a hash table has enough collisions, it is split and added to the tree, without affecting the other hash tables. Clients use notifications to learn that a tree node has changed; alternatively, we allow client caches to become stale and introduce a version number for each hash table, kept at the client trees and each item in the data structure in far memory. When a client accesses an item, it checks that its cached version matches what is in far memory, to learn whether its cache is stale. When the cache is up-to-date, clients look up items in one far access, and store items in two (one access checks the version prior to updating a bucket pointer with a CAS). An HT-tree can store 1 trillion items with a tree of 10M nodes (taking 100s of MB of cache space) and 10M hash tables of 100K elements each.

5.3 Queues

Queues are useful to store and consume work items. A queue is typically implemented as a large array, a head pointer, and a tail pointer. Items are added to the tail, and they are removed from the head. Races are a challenge: when many clients operate on the queue concurrently (updating the head, tail, and the array simultaneously), they could corrupt the queue.

We address this problem by using fetch-and-add-indirect and store-and-add-indirect (*faai*, *saai*). These instructions permit a client to do two things atomically: (1) update the head or tail pointers and (2) extract or insert the required item. As a result, we can execute dequeue and enqueue operations without costly concurrency control mechanisms (e.g., locks, lock-free techniques), with one far access in the common *fast-path* case, which is efficient. Infrequent corner cases trigger a *slow-path*, which executes less efficiently, requiring additional far accesses. There are two corner cases: when the head and tail pointers wrap around, and when the queue is empty or near empty. Clients must determine that they should run the slow-path without incurring additional far accesses in the fast-path. To do this, we allocate a slack region past the array consisting of $n + 1$ locations, where n is a bound on the number of clients. Clients check if they reach the slack after an operation completes, in the background, so they need not check during the fast-path whether the head or tail requires a wrap around. Similarly, clients do not check if the queue is full or empty during the fast-path, relying on a (second) logical slack region to keep the head and tail $2n$ positions apart. (Due to space constraints, we omit the details here.)

5.4 Refreshable vectors

Caching a vector at clients (§5.1) may generate excessive notifications when the vector changes often. To address this issue, we propose *refreshable vectors*, which can return stale data, but include a refresh operation to guarantee the freshness of the next lookup. This abstraction is useful in distributed machine learning to store model parameters [27]: workers read

parameters from the vector and refresh periodically to provide bounded staleness and guarantee learning convergence.

To implement refreshable vectors, each client keeps a cached copy of the vector. We optimize refresh so that clients need not read the full vector, applying a dynamic policy that shifts from client-initiated version checks (when data is changing frequently) to a notification-based scheme as the rate of updates slows. Vector entries are grouped, with a version number per group; a client reads the version numbers from far memory, compares against its cached versions, and then uses a gather operation (*rgather*) to read at once all entries of groups whose versions have changed. To avoid the latency of explicitly reading slowly changing version numbers from far memory as iterative algorithms converge, a client can use a version number invalidation (*notify0*) to learn when any element of the corresponding group has changed. Changes in per-group version numbers will prompt the client to refresh the entire group, even if only a subset of the elements has changed. If, instead, individual vector element version numbers are tracked in a contiguous memory region, the client can use an update notification (*notify0d*) for the version number region to learn which specific entries have changed.

6 Case study: Monitoring

We now describe how to apply our ideas to a monitoring application, where a sampled metric (e.g., CPU utilization) is tracked in far memory. The system must raise alarms of different severity if the samples exceed some predefined thresholds (warning, critical, failure) for a certain duration within a *time window* (e.g., 10s). The clients of the monitoring service are a producer and multiple consumers. In a naive implementation, the producer writes the metric samples to far memory, and consumers read the data for analysis. Each sample is written once and read by all consumers, resulting in $(k + 1)N$ far memory transfers for N samples and k consumers.

With our proposed hardware primitives and data structures, we can do much better. Rather than storing samples, far memory keeps a vector (§5) with a histogram of the samples. The producer treats a sample as an offset into the vector, and increments the location using one far memory access with indexed indirect addressing (Fig.1). Each consumer uses notifications (Fig.1) to get changes in the histogram vector at offsets corresponding to the alarm ranges. Since the samples are often in the normal range, notifications are rare, reducing far memory transfers from N to $m<<N$. Consumers optionally copy (with an extra far memory access) the histogram values in the prescribed range for further aggregation.

The above works if we assume a single time window. To track multiple windows, we can use a collection of histogram vectors implemented as a circular buffer, with a base pointer to the current vector (§5). After a window ends, the producer switches the base pointer in far memory and the client is notified (Fig.1).

With this approach, the producer and consumers operate independently. Different consumers can be notified of different thresholds and take different actions. Only exceptional events cause data transfers to the consumer, reducing far memory traffic. Since consumers can access the distribution over a number of windows, they can also correlate the histograms to detect variations in the metric over multiple windows. While simple, this use case shows how far memory can be used as an intermediary to reduce interconnect traffic.

7 Implementation scalability

Implementing our proposed hardware primitives requires addressing scalability in the size of far memory, the number of clients, and the expected usage patterns.

7.1 Indirect addressing in large far memories

Large far memories comprise many memory nodes, with the far memory address space distributed across these nodes. Similar to interleaving in traditional local memories, data may be striped across multiple far memory nodes to increase aggregate far memory bandwidth and parallelism. This architecture presents challenges for memory-side indirect addressing: a dereferenced pointer may refer to data held by a remote memory node. If the target memory node can be easily determined from the address, then the memory node processing the indirection may forward the request to the memory node storing the dereferenced pointer target. Alternately, a request for a dereferenced remote address could return an error, indicating only direct addressing, and leaving it up to the compute node to explicitly issue a request to the target memory node to complete the indirection. Note that both cases require more than a single round trip to far memory, with request forwarding performing fewer network traversals.

In addition to this data-unaware approach to far memory interleaving, data may be distributed across far memory in a data structure-aware fashion (e.g., for higher levels of a tree, or for very large keys or values), to provide more parallelism between independent requests. Parts of the data structure where indirect addressing is expected to be common (e.g., a chain within a hash bucket) may benefit from localized placement, to minimize network traversals for indirection. Far memory allocators may be designed with locality in mind, to permit applications to provide hints about the desired (anti-)locality of a data structure, which the allocator can consider when granting the allocation request.

7.2 Notification scalability

Notifications require keeping track of which clients want to be notified (the subscribers) for which memory ranges (the subscriptions). A hardware implementation of notifications faces several scalability challenges:

Number of subscribers. To scale, we use a software-hardware co-design: the subscribers of the hardware primitives (Fig. 1) are compute nodes, and a software layer on each compute node

routes notifications to individual processes. We can also use a publish-subscribe architecture [13]: the hardware subscribers are dedicated software brokers (10–100s of them), which then route notifications to the subscribers over the network.

Number of subscriptions. To scale, we can increase the spatial granularity of the hardware subscriptions (e.g., two subscriptions on nearby ranges become one subscription on an encompassing range). An update would trigger a notification for the encompassing range, leading to potential false positives for the original subscriptions, which the subscriber would need to check. Alternatively, the notification can carry enough information about the triggering update to allow a software layer to distinguish between the original ranges.

Network traffic. Depending on application consistency goals, we can coalesce many notifications to the same subscription (i.e., temporal batching). During traffic spikes, we can drop notifications for entire periods (e.g., seconds), replacing them with a warning that notifications were lost. The data structure algorithm then adapts accordingly, based on the desired consistency goals.

8 Related work

Data structures and algorithms. Prior work proposes methods to build better data structures using one-sided accesses. DrTM+H [35] caches hash table entry addresses on the client for later reuse. FaRM [11] uses Hopscotch hashing, where multiple colliding key-value pairs are inlined in neighboring buckets, allowing clients to read multiple related items at once. These approaches have drawbacks: DrTM+H keeps significant metadata on clients, while FaRM consumes additional bandwidth to transfer items that will not be used. The Berkeley Container Library (BCL) [6] focuses on avoiding coordination between clients and building a usable library for applications. Other authors [12] consider B-trees with one-sided access. Several papers argue against data structures with one-sided access [24, 25]. What distinguishes our proposal from all of the above is an emphasis on designing fundamentally new data structures that reduce far memory access to a minimum—one far access most of the time—which we find to be a requirement to compete with distributed data structures based on RPCs.

The goal of avoiding far memory accesses is an example of a broader class of communication-avoiding algorithms [10], which aim to avoid communication (i.e., moving data between memory hierarchy levels and between processors over a network) because it is more costly than arithmetic computation. Follow-on work in write-avoiding algorithms aims to avoid the penalties of asymmetric write performance [4, 8] in emerging non-volatile storage devices. This work designs algorithms to avoid expensive operations, and, where possible, achieve lower bounds on the number of (reads and) writes.

Our HT-tree is a data structure that combines multiple data structures (a tree of hash tables). This idea has been proposed with other data structures, to achieve objectives different from ours. MassTree [31] uses a trie of B+trees. A burst trie [17] is a trie with each leaf node replaced by a small container data structure (linked list, binary search tree, or splay tree). In particular, a HAT-trie [3] uses an array as the container. The goal of burst- and HAT-tries is efficient storage of variable-length strings.

Distributed shared memory. Far memory data structures permit different data structures to use different strategies for maintaining their consistency. An early application of this idea was in the Munin distributed shared memory (DSM) system [9], which used differentiated consistency policies for different data structures, based on sharing patterns. Variable declarations are annotated to indicate sharing patterns, with system- and user-defined consistency protocols used to enforce desired consistency levels while minimizing the communication needed to keep the distributed memories consistent.

GAM [7] is a recent distributed shared memory system implementing directory-based cache coherence over RDMA. GAM executes writes asynchronously and allows them to bypass other reads and writes, providing partial store order (PSO) consistency. Unreliable notifications in far data structures provide even weaker guarantees, as invalidations upon writes not only may arrive out of order (due to network congestion), but may never arrive. With weak consistency models, the programmer needs to take additional care (e.g., using fences in PSO) to ensure correctness. In far data structures, high-priority warning messages and versioning serve as helper mechanisms to the programmer, compensating for the lack of reliable and timely notifications.

Hardware support. Hardware changes for data structures have been proposed before, such as pointer chasing in the context of 3D die-stacked memory and FPGAs [20, 36]. Both proposals require complex support for traversing pointer-linked data structures of arbitrary length. Instead, we look for small hardware extensions (i.e., no loops) for indirect addressing, scatter-gather, and notifications.

Protozoa is an adaptive granularity cache coherence proposal enabling data movements to be matched with an application's spatial locality [37]. Unlike Protozoa, far memory data structures propose variable coherence notification subscriptions for the purpose of reducing the amount of metadata.

9 Conclusion

Technologies like RDMA and Gen-Z extend memory beyond what is in the box, to a large far memory. However, existing data structures are poorly suited for far memory, due to the one-sided model used to access it. We introduced far memory data structures to address this problem; these new data structures reduce far accesses with the help of simple hardware extensions to far memory. This combination of software and hardware techniques will enable programmers to *think outside the box*.

References

[1] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *USENIX Annual Technical Conference, (ATC)*, pages 775–787, July 2018.

[2] K. Asanović. FireBox: A hardware building block for 2020 warehouse-scale computers. In *USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2014.

[3] N. Askitis and R. Sinha. HAT-trie: A cache-conscious trie-based data structure for strings. In *Australasian Conference on Computer Science (ACSC)*, pages 97–105, Jan. 2007.

[4] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–156, July 2016.

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

[6] B. Brock, A. Buluç, and K. A. Yelick. BCL: A cross-platform distributed container library. *CoRR*, abs/1810.13029, Oct. 2018.

[7] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K. Tan, Y. M. Teo, and S. Wang. Efficient distributed memory management with RDMA and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, July 2018.

[8] E. Carson, J. Demmel, L. Grigori, N. Knight, P. Koanantakool, O. Schwartz, and H. V. Simhadri. Write-avoiding algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 648–658, May 2016.

[9] J. B. Carter. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29(2):219–227, Sept. 1995.

[10] J. Demmel. Tutorial: Introduction to communication-avoiding algorithms. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2016.

[11] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, Apr. 2014.

[12] A. Dragojevic, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 54–70, Oct. 2015.

[13] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[14] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic. Beyond processor-centric operating systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2015.

[15] Gen-Z Core Specification, Revision 1.0. http://www.genzconsortium.org.

[16] Gen-Z Atomics. http://genzconsortium.org/wp-content/uploads/2017/08/Gen-Z-Atomics.pdf, July 2017.

[17] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223, Apr. 2002.

[18] Hewlett Packard Enterprise The Machine. https://www.labs.hpe.com/the-machine.

[19] High throughput computing data center architecture. http://www.huawei.com/ilink/en/download/HW_349607.

[20] K. Hsieh, S. M. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *IEEE International Conference on Computer Design (ICCD)*, pages 25–32, Oct. 2016.

[21] Intel Omni-Path Architecture. http://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html.

[22] Intel Rack Scale Design. http://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[23] Intel/Facebook Disaggregated Rack. http://goo.gl/6h2Ut.

[24] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM*, pages 295–304, Aug. 2014.

[25] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 185–201, Nov. 2016.

[26] K. Keeton, S. Singhal, and M. Raymond. The OpenFAM API: A programming model for disaggregated persistent memory. *OpenSHMEM and Related Technologies: OpenSHMEM in the Era of Extreme Heterogeneity, Springer Lecture Notes in Computer Science*, 11283:70–89, Mar. 2019.

[27] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 583–598, Oct. 2014.

[28] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *International Symposium on Computer Architecture (ISCA)*, pages 267–278, June 2009.

[29] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Feb. 2012.

[30] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the design and scalability of distributed shared-data databases. In *International Conference on Management of Data (SIGMOD)*, pages 663–676, May 2015.

[31] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *ACM European Conference on Computer Systems (EuroSys)*, pages 183–196, Apr. 2012.

[32] S. Novakovic, A. Daglis, D. Ustiugov, E. Bugnion, B. Falsafi, and B. Grot. Mitigating load imbalance in distributed data serving with rack-scale memory pooling. *ACM Transactions on Computer Systems*, 36(2), Apr. 2019.

[33] RDMA Consortium. http://www.rdmaconsortium.org/.

[34] H. Shah, F. Marti, W. Noureddine, A. Eiriksson, and R. Sharp. Remote direct memory access (RDMA) protocol extensions. https://tools.ietf.org/html/rfc7306, June 2014.

[35] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 233–251, Oct. 2018.

[36] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe. A study of pointer-chasing performance on shared-memory processor-FPGA systems. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 264–273, Feb. 2016.

[37] H. Zhao, A. Shriraman, S. Kumar, and S. Dwarkadas. Protozoa: adaptive granularity cache coherence. In *International Symposium on Computer Architecture (ISCA)*, pages 547–558, June 2013.