AIFM: High-Performance, Application-Integrated Far Memory

Zhenyuan Ruan, *MIT CSAIL;* Malte Schwarzkopf, *Brown University;*
Marcos K. Aguilera, *VMware Research;* Adam Belay, *MIT CSAIL*

https://www.usenix.org/conference/osdi20/presentation/ruan

This paper is included in the Proceedings of the
14th USENIX Symposium on Operating Systems
Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the
14th USENIX Symposium on Operating
Systems Design and Implementation
is sponsored by USENIX

AIFM: High-Performance, Application-Integrated Far Memory

Zhenyuan Ruan Malte Schwarzkopf † Marcos K. Aguilera ‡ Adam Belay
MIT CSAIL *†Brown University* *‡VMware Research*

Abstract. Memory is the most contended and least elastic resource in datacenter servers today. Applications can use only local memory—which may be scarce—even though memory might be readily available on another server. This leads to unnecessary killings of workloads under memory pressure and reduces effective server utilization.

We present *application-integrated far memory* (AIFM), which makes remote, "far" memory available to applications through a simple API and with high performance. AIFM achieves the same common-case access latency for far memory as for local RAM; it avoids read and write amplification that paging-based approaches suffer; it allows data structure engineers to build *remoteable*, hybrid near/far memory data structures; and it makes far memory transparent and easy to use for application developers.

Our key insight is that exposing application-level semantics to a high-performance runtime makes efficient remoteable memory possible. Developers use AIFM's APIs to make allocations remoteable, and AIFM's runtime handles swapping objects in and out, prefetching, and memory evacuation.

We evaluate AIFM with a prototypical web application frontend, a NYC taxi data analytics workload, a memcached-like key-value cache, and Snappy compression. Adding AIFM remoteable memory to these applications increases their available memory without performance penalty. AIFM outperforms Fastswap, a state-of-the-art kernel-integrated, paging-based far memory system [6] by up to 61×.

1 Introduction

Memory (RAM) is the most constrained resource in today's datacenters. For example, the average memory utilization on servers at Google [73] and Alibaba [46] is 60%, with substantial variance across servers, compared to an average CPU utilization of around 40%. But memory is also the most inelastic resource: once a server runs out of available memory, some running applications must be killed. In a month, 790k jobs at Google had at least one instance killed, in many cases due to memory pressure [73]. A killed instance's work and accumulated state are lost, wasting both time and energy. This waste happens even though memory may be available on other servers in the cluster, or even locally: around 30% of server memory are "cold" and have not been accessed for minutes [41], suggesting they could be reclaimed.

Operating systems today support memory elasticity primarily through *swap* mechanisms, which free up RAM by pushing unused physical memory pages to a slower tier of memory, such as disks or remote memory. But OS swap mechanisms operate at a fixed and coarse granularity and incur substantial overheads. To swap in a page, the OS must handle a page fault, which requires entering the kernel and waiting until the data arrives. Figure 1 shows the throughput a recent page-based far memory system (*viz.*, Fastswap [6]) achieves when accessing remote objects using up to four CPU cores. Kernel swapping happens at the granularity of 4KB pages, so page-based far memory suffers read/write amplification when accessing small objects, as at least 4KB must always be transferred. Moreover, the Linux kernel spins while waiting for data from swap to avoid the overheads of context switch and interrupt handling. That means the wait time (about 15–20k cycles with Fastswap's RDMA backend) is wasted.

Throughput [accesses/sec]	*64B object*	*4KB object*
Paging-based (Fastswap [6])	582K	582K
AIFM	3,975K	1,059K

Figure 1: AIFM achieves 6.8× higher throughput for 64B objects and 1.81× higher throughput for 4KB objects, compared to Fastswap [6], a page-granular, kernel-integrated far memory approach. AIFM performs well since it (*i*) avoids IO amplification and (*ii*) context switches while waiting for data.

We describe a fundamentally different approach: *application-integrated far memory* (AIFM), which ties swapping to individual application-level memory objects, rather than the virtual memory (VM) abstraction of pages. Developers write *remoteable* data structures whose backing memory can be local and "far"—*i.e.*, on a remote server—without affecting common-case latency or application throughput. When AIFM detects memory pressure, its runtime swaps out objects and turns all pointers to the objects into remote pointers. When the application dereferences a remote pointer, a lightweight green threads runtime restores the object to local memory. The runtime's low context switch cost permits other green threads to make productive use of the wait cycles, which hides remote access latency and maintains high throughput. Due to these fast context switches, AIFM achieves 81% higher throughput than page-based approaches when accessing 4KB objects, and because AIFM avoids amplification, it achieves 6.8× higher throughput for small objects (Figure 1).

AIFM's programming interface is based on four key ideas: a fast, low-overhead *remoteable pointer* abstraction, a pauseless memory evacuator, runtime APIs that allow data struc-

tures to convey semantic information to the runtime, and a *remote device* interface that helps offload light computations to remote memory. These AIFM APIs allow data structure engineers to build hybrid local/remote data structures with ease, and provide a developer experience similar to C++ standard library data structures. The pauseless memory evacuator ensures that application threads never experience latency spikes due to swapping. Because data structures convey their semantics to the runtime, AIFM supports custom prefetching and caching policies—*e.g.*, prefetching remote data in a remoteable list and streaming of remote data that avoids polluting the local memory cache. Finally, AIFM's offloading reduces data movement and alleviates the network bottleneck that most far-memory systems experience.

The combination of these ideas allows AIFM to achieve object access latencies bounded only by hardware speed: if an object is local, its access latency is comparable to an ordinary pointer dereference; when it is remote, AIFM's access latency is close to the hardware device latency.

We evaluate AIFM with a real-world data analytics workload built on DataFrames [16], a synthetic web application frontend that uses several remoteable data structures, as well as a memcached-style workload, Snappy compression, and microbenchmarks. Our experiments show that AIFM maintains high application request throughput and outperforms a state-of-them-art, page-based remote memory system, Fastswap, by up to 61×. In summary, our contributions are:

1. Application-integrated far memory (AIFM), a new design to extend a server's effective memory size using "far" memory on other servers or storage devices.
2. A realization of AIFM with convenient APIs for development of applications and remoteable data structures.
3. A high-performance runtime design using green threads and a pauseless memory evacuator that imposes minimal overhead on local object accesses and avoids wasting cycles while waiting for remote object data.
4. Evaluation of our AIFM prototype on several workloads, and microbenchmarks that justify our design choices.

Our prototype is limited to unshared far memory objects on a single memory server. Future work may add multi-server support, devise strategies for dynamic sizing of remote memory, or investigate sharing.

2 Background and Related Work

OS swapping and far memory. Operating systems today primarily achieve memory elasticity by *swapping* physical memory pages out into secondary storage. Classically, secondary storage consisted of disks, which are larger and cheaper but slower than DRAM. The use of disk-based swap has been rare in datacenters, since it incurs a large performance penalty. More recent efforts consider swapping to a faster tier of memory or *far memory*, such as the remote memory of a host [3, 6, 21, 27, 28, 31, 40, 45, 48, 67] or a compression cache [24, 41, 81, 82]. Since swapping is integrated with the kernel virtual memory subsystem, it is transparent to user-space applications. But this transparency also forces swapping granularity to the smallest virtual memory primitive, a 4KB page. Combined with memory objects smaller than 4KB, this leads to *I/O amplification*: when accessing an object, the kernel must swap in a full 4KB page independent of the object's actual memory size. Moreover, supplying application semantic information, such as the expected memory access pattern, the appropriate prefetch strategy, or memory hotness, is limited to coarse and inflexible interfaces like `madvise`.

AIFM uses far memory in a different way from swapping, by operating at *object granularity* rather than page-granularity—an idea that we borrow from prior work on distributed shared memory (see below), memory compression [75], and SSD storage [1]. These investigations all point to page-level I/O amplification as a key motivation.

AIFM provides transparent access to far memory using smart pointers and dereference scopes inspired by C++ weak pointers [69], and Folly RCU guards [26].

Disaggregated and distributed shared memory. Disaggregated memory [58] refers to a hardware architecture where a fast fabric connects hosts to a pool of memory [29, 33], which is possibly managed by a cluster-wide operating system [33, 66]. Disaggregated memory requires new hardware that has not yet made it to production. AIFM focuses on software solutions for today's hardware.

Distributed shared memory (DSM) provides an abstraction of shared memory implemented over message passing [7, 10, 44, 50, 64, 65]. Like far memory, DSM systems can be page-based or object-based. DSM differs from far memory both conceptually and practically. Conceptually, DSM provides a different abstraction, where data is *shared* across different hosts (the "S" in DSM). Practically, this abstraction leads to complexity and inefficiency, as DSM requires a cache coherence protocol that impairs performance. For instance, accessing data must determine if a remote cache holds a copy of the data. By contrast, data in far memory is private to a host—a stricter abstraction that makes it possible to realize far memory more efficiently. Finally, DSM systems were designed decades ago, and architectural details and constants of modern hardware differ from their environments.

Technologies to access remote data. TCP/IP is the dominant protocol for accessing data remotely, and AIFM currently uses TCP/IP. Faster alternatives to TCP/IP exist, and could be used to improve AIFM further, but these technologies are orthogonal or complementary to AIFM's key ideas.

RDMA is an old technology that has recently been commoditized over Ethernet [32], generating new interest. Much work is devoted to using RDMA efficiently in general [39, 51, 76] or for specific applications, such as key-value stores (*e.g.*, [38, 49]) or database systems [11]. Smart NICs use CPUs or FPGAs [47, 52, 70] to provide programmable remote functionality [18, 43, 68]. AIFM requires no specialized hardware.

Abstractions for remote data. Remote Procedure Calls (RPCs) [12] are widely used to access remote data, including over RDMA [19, 71] or TCP/IP [37]. Memory-mapped files can offer remote memory behind a familiar abstraction [2, 67], while data structure libraries for remote data [4, 15], offer maps, sets, multisets, lists, and other familiar constructs to developers. This is similar in spirit to data structure libraries for persistent memory [59, 62]. AIFM offers a lower-level service that helps programmers develop such data structures.

I/O amplification. As mentioned, page-based access leads to I/O amplification, a problem studied extensively in the context of storage systems [1, 61] and far-memory systems [17], where hardware-based solutions can reduce amplification by tracking accesses at the granularity of cache lines.

Garbage collection and memory evacuation. Moving objects to remote memory in AIFM ("evacuation") is closely related to mark-compact garbage collection (GC) in managed languages. The main difference is that AIFM aims to increase memory capacity by moving cold, but live objects to remote memory, while GCs focus on releasing dead, unreferenced objects' memory. AIFM uses referencing counting to free dead objects, avoiding the need for a tracing stage. Instead of inventing a new evacuation algorithm, AIFM borrows ideas from the GC literature and adapts them to far-memory systems. Like GCs, AIFM leverages a read/write barrier to maintain object hotness [5, 14, 34], but AIFM uses a one-byte hotness counter instead of a one-bit flag, allowing more fine-grained replacement policies. Like AIFM, some copying collectors optimize data locality by separating hot and cold data during GC, but target different memory hierarchies; *e.g.*, the cache-DRAM hierarchy [34], the DRAM-NVM hierarchy [5, 79, 80], and the DRAM-disk hierarchy [14]. Finally, memory evacuation interferes with user tasks and impacts their performance. To reduce the interference, AIFM adopts an approach similar to the pauseless GC algorithms in managed languages [20], as opposed to the stop-the-world GC algorithms [36].

3 Motivation

Kernel paging mechanisms impose substantial overheads over the fundamental cost of accessing far memory.

Consider Figure 2, which breaks down the costs of Linux (v5.0.0) retrieving a swapped-out page from an SSD. The device's hardware latency is about 6µs, but Linux takes over 15µs (2.5×) due to overheads associated with locking (P1, P5), virtual memory management (P2, P3, P5), accounting (P4), and read IO amplification (P3). Moreover, due to the high cost of context switches, Linux spins while waiting for data (P3), wasting 11.7µs of possible compute time.

AIFM, by contrast, provides low-overhead abstractions and an efficient user-space runtime that avoid these costs, bringing its latency (6.8µs) close to the hardware limit of 6µs. We explain these concepts in the next two sections.

Phase	Linux Kernel Swapping	AIFM
P1	Page fault, trap to kernel	Deref far pointer, issue I/O
P2	Lock, get PTE, allocate page frame, allocate swap cache entry	Lightweight context-switch
P3	Issue read I/O, spin, insert PFN in global LRU list	Run another green thread
P4	cgroup accounting, reclaim memory if past limit	I/O completion, context-switch back
P5	Set page mapping, unlock	—

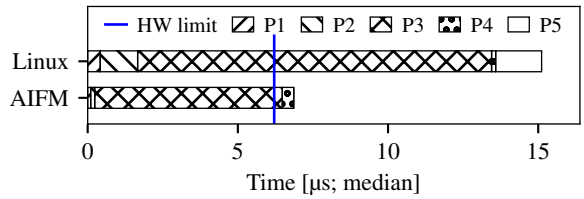


Figure 2: Linux kernel-based swapping has high overheads over hardware I/O limits (blue line, 6µs). Both Linux and AIFM use an SSD device backend in this experiment.

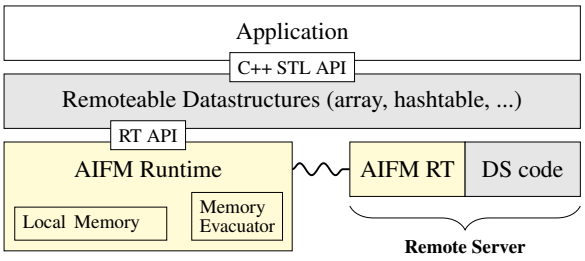


Figure 3: Applications use remoteable data structures (gray), and data structure developers rely on the AIFM runtime (yellow) to handle local memory management and interact with remote memory. Data structures can have active remote components (*i.e.*, the "DS code" box) to offload light computation.

4 AIFM Design

The goal of Application-Integrated Far Memory (AIFM) is to provide an easy-to-use, efficient interface for far memory without the overheads of page-granular far memory.

4.1 Overview

AIFM targets two constituencies: application developers and data structure developers. AIFM provides application developers with data structures with familiar APIs, allowing developers to treat these *remoteable* data structures mostly as black boxes; and AIFM provides simple, but powerful APIs to data structure engineers, allowing them to implement a variety of efficient remoteable memory data structures. Figure 3 shows a high-level overview of AIFM's design: applications interact with data structures (gray) implemented using primitives and APIs provided by the AIFM runtime (yellow).

For an **application developer**, programming applications that use far memory should feel almost the same as programming with purely local data structures. In particular, the developer should not need to be aware of whether an object is currently local or remote (*i.e.*, far memory is *transparent*), and remoteable memory data structures should offer the same

performance as local ones in the common case. For example, idiomatic C++ code for reading several hash table entries and an array element computed from them might look as follows:

```
std::unordered_map<key_t, int> hashtable;
std::array<data_t> arr;

void print_data(std::vector<key_t>& request_keys) {
  int sum = 0;
  for (auto key : request_keys) {
    sum += hashtable.at(key);
  }
  std::cout << arr.at(sum) << std::endl;
}
```

The same code written using AIFM looks like this:

```
RemHashtable<key_t, int> hashtable;
RemArray<data_t> arr;

void print_data(std::vector<key_t>& request_keys) {
  int sum = 0;
  for (auto key : request_keys) {
    DerefScope s1; // Explained in Section 4.2.2.
    sum += hashtable.at(key, s1);
  }
  DerefScope s2;
  std::cout << arr.at(sum, s2) << std::endl;
}
```

The remoteable memory data structures themselves (`RemHashtable` and `RemArray` above) are written by **data structure engineers**, who use AIFM's runtime APIs to include remoteable memory objects in their data structures. When memory becomes tight, AIFM's runtime moves some of these memory objects to remote memory; when the data structure needs to access remote objects, the AIFM runtime fetches them. Data structure engineers have substantial design freedom: they can rely entirely on AIFM to fetch remote objects, or they can deploy custom logic on the remote side.

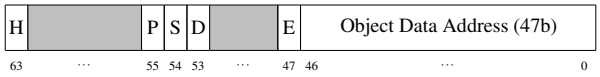
Remote servers store the actual remote data in their memory, and run a counterpart AIFM runtime, which may call into custom data structure logic. This is helpful, *e.g.*, if the remoteable memory data structure needs to chase pointers, which would otherwise require multiple round-trips.

4.2 Remoteable Memory Abstractions

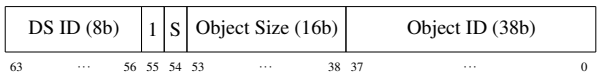
AIFM is designed around four core abstractions: *remoteable pointers*, *dereference scopes*, *evacuation handlers*, and *remote devices*. We designed the abstractions such that they impose minimal overheads (as low as three micro-ops) on "hot path" access to local objects, and try to ensure that the "cold path" remote access incurs little latency above hardware limits.

4.2.1 Remoteable Pointers

A remoteable pointer represents a memory object (*i.e.*, an allocation) that is currently either local, or remote (in "far" memory). AIFM supports unique and shared remoteable pointers, whose interface makes them suitable for use in any place where a data structure would use an ordinary, local pointer.



(a) Local object. H: hot, P: present, S: shared, D: dirty, E: evacuating.



(b) Remote (swapped-out) object. DS ID means data structure ID.

Figure 4: Remoteable unique pointer representations for local and remote objects. AIFM inverts the H/P/D bit meaning (0 = hot/present/dirty) for a more efficient hot path execution.

Memory representation. Unique remoteable pointers, which correspond to C++'s `std::unique_ptr`, have the same size as ordinary 64-bit pointers, while shared pointers are 128-bits wide (like `std::shared_ptr`). Figure 4 shows the memory layout of a remoteable unique pointer. Depending on whether a remoteable pointer is local or remote, we adopt a different format. If the memory is local (Figure 4a), the pointer contains a virtual memory address in its lower 47 bits (enough to represent user-space addresses), and control bits in the upper 17 bits, including standard dirty (D) and present (P) bits (cf. page tables). It also contains bits to track whether the pointer is hot (H) and whether it is being concurrently evacuated (E). For unique pointers, the shared (S) bit is set to 0. We byte-align the D, E, and H bits, allowing each of them to be accessed by mutators and runtime evacuators concurrently and atomically, as a byte is the smallest read/write unit.

If the memory is remote (Figure 4b), it contains metadata to assist in retrieving the object from remote memory, such as the data structure ID, the object size, and the object ID. Each data structure instance has a unique data structure ID managed by the runtime. The object ID refers to a data structure-specific object identifier (such as a key in a hash table), which is used by the remote memory server to identify the object.

AIFM's remoteable shared pointer, which allows pointer aliasing and corresponds to C++'s `std::shared_ptr` differs from the unique pointer in two ways. First, its S bit is set to 1; and second, the pointer has an additional 8 bytes for chaining the shared pointers to the same object. When AIFM's runtime evacuates the referred object or moves it locally (§5.3), it traverses the chain to update all shared pointers.

API. Listing 1 shows the API of the remoteable unique pointer (the shared pointer's API is largely identical). `RemUniquePtr` has two constructors: one for already-local objects and one for currently remote objects. The second constructor allows data structures to form remoteable pointers to objects that are currently remote. This helps data structure engineers reference remote objects from their data structures without having to fetch those objects.

To turn a remote pointer into a local one, the programmer dereferences it via the `deref` and `deref_mut` API methods.

```
class RemUniquePtr<T> {
  uint64_t metadata; // 64 bits, see Figure 4.
  // Construct local object
  RemUniquePtr(DSID, T* obj_addr);
  // Construct remote object
  RemUniquePtr(DSID, ObjID);
  const T* deref(DerefScope& scope); // Immutable.
  T* deref_mut(DerefScope& scope); // Mutable.
}
```

Listing 1: AIFM remoteable unique pointer API.

Dereferencing. When the dereferencing methods are called, the runtime inspects the present bit of the remoteable pointer. If the object is local, it sets the hot bit and returns the address stored in the pointer. Otherwise, the runtime fetches it from the remote server, sets the hot bit and dirty bit (in `deref_mut`), and returns a local pointer to the data.

AIFM's hot path for local access is carefully optimized and takes five x86-64 machine instructions: one `mov` to load the pointer, one `andl` to check present and evacuating bits, a conditional branch to the cold path if neither is set, a shift (`shrq`) to extract the object address, and a `mov` to return it. Modern x86-64 processors macro-fuse the second and third instructions (test and branch), so the hot path requires four micro-ops, a three-micro-op overhead over an ordinary pointer dereference. The cold path is slower, as it calls into the AIFM runtime to potentially swap in a remote object.

One challenge to making this API work is managing the local lifetime of the dereferenced data: while the application holds a pointer returned from dereferencing a `RemUniquePtr`, the runtime must never swap out the object. This is hard to achieve in unmanaged languages like C/C++, since after getting the raw address, application code could store it virtually anywhere (*e.g.*, on the heap, stack, or even in registers). The runtime lacks sufficient information to detect whether any such pointer continues to exist, and thus whether the data is still being used. The Boehm garbage collector [13] tackles a similar reference lifetime problem by scanning the whole address space to find any possible references. Such scans would impose an unacceptable performance overhead for AIFM. Our solution is to instead leverage application semantics to tie the lifetime of the local, dereferenced data to the lifetime of the AIFM's dereference scopes.

4.2.2 Dereference Scopes

Listing 2 demonstrates the usage of `DerefScope`. Before accessing the remoteable object, the developer must construct a `DerefScope`. AIFM container's API provides a compile-time check by taking a `DerefScope&` argument. (This is also why the remoteable pointer has its own dereferencing methods, rather than overloading `operator*`.)

Under the hood, `DerefScope`'s constructor creates an evacuation fence, which blocks upcoming evacuations until it is destructed. The lifetime of all local dereferenced data is therefore tied to the scope lifetime. Accessing dereferenced data

```
RemVector<value_t> vec;
// ...
for (uint64_t i = 0; i < vec.size(); i++) {
  {
    DerefScope scope;
    auto& value = vec->at(i, scope);
    // process value
  }
  // scope destroyed, can evacuate value's object
}
```

Listing 2: AIFM dereference scope example.

outside the dereference scope is undefined behavior. In the future, AIFM might leverage static analysis to catch lifetime violations, as in the Rust compiler [78].

Our scope API is familiar to C/C++ programmers; it shares similarity with C++11's `std::weak_ptr` and, *e.g.*, the `rcu_reader` guard in Facebook's RCU API [26]. Note that the lifetime of the `DerefScope` is separate from the lifetime of the remoteable pointer: a remoteable pointer may still be alive even when its data has been swapped to the remote. This is unlike, *e.g.*, `std::unique_ptr`, where the pointer's destructor terminates the lifetime of the object data.

Dereference scopes require developers to modify the application code. An alternative API might avoid the need for a dereference scope at the cost of copying the object into local memory on dereference. AIFM's core APIs aim to achieve maximum performance, so we avoid copying by default. The overhead of a copying API is highly application-dependent; our experiments suggest that 3–8% overhead are typical for applications with high compute/memory access ratios.

4.2.3 Evacuation Handlers

When an object is not protected by a `DerefScope`, AIFM's runtime may evacuate it to far memory. Evacuation changes the pointer to this object from local to remote status, and future dereferences will cause AIFM to swap the object back in. But some use cases may wish to implement custom behavior on evacuation. For example, when AIFM evacuates an object contained in a hash table, the hash table may register an evacuation handler to remove the key and object pointer to save local space. (In this case, future lookup misses for the key will reconstitute the key and pointer, and add them to the hash table.) AIFM offers *evacuation handlers* for this purpose, enabling developers to incorporate the data structure semantics into the runtime evacuator.

Evacuation handlers are also critical for handling embedded remoteable pointers inside objects. For example, data structure engineers can use evacuation handlers to support embedded remoteable unique pointers in objects that are themselves remoteable. When an object is remoted, any embedded remoteable pointers must either be moved to the local heap, or the object it references must be moved to remote memory, and the remoteable pointer must be updated with an identifier to later retrieve the remote object from a remote device (§4.2.4). As a result, the evacuator never has to retrieve remote memory

```
// The AIFM runtime will invoke the handler on evacuating
// the object to the remote server (phase 4 in Section 5.3).
using EvacHandler = std::function<
    void(Object&, const Runtime::CopyToRemoteFn&)>;
// Registers an evacuation handler for a data structure ID.
void Runtime::RegisterEvacHandler(DSID id, EvacHandler h);
```

Listing 3: AIFM evacuation handler API.

to update a remoteable pointer.

AIFM provides an evacuation handler API (Listing 3). The evacuation handler gets invoked on evacuating the object to the remote server (phase 4 in §5.3), right before the runtime frees the object's local memory. The runtime passes two arguments to `EvacHandler`—the object to be evacuated and the function that triggers the runtime to copy the object to the remote side. The first argument allows the handler to mutate the object data before copying (*e.g.*, modify the state of its embedded pointers) and further cleanup the local data structure after copying (*e.g.*, remove its pointer from the hash table index). The second argument offers the flexibility in the timing of copying the object to remote.

Data structure developers register their evacuation handlers by invoking `RegisterEvacHandler`. An evacuation handler is tied to a unique data structure ID, which each data structure allocates in its constructor, and which data structure engineers must use consistently. This way, different data structures or instances of the same data structure coexist in the same application, while the runtime invokes the appropriate handler.

4.2.4 Remote Devices

AIFM's `RemDevice` provides functionality at the remote memory server (Listing 4). The remote device, by default, uses a key-value store abstraction: when the client dereferences a remote pointer, the runtime sends the data structure ID and object ID to the remote server, which looks up the object by data structure ID and object ID, and sends the object data back. When evacuating an object, the runtime sends IDs and object data to the remote server, which inserts the object.

AIFM also gives datastructure engineers the flexibility to override this default behavior to integrate custom active components at the remote server. This is accomplished by registering their implementation on their own data structure type to the remote device (`register_active_component`). A custom active component is especially beneficial when the application's compute intensity is low, as this setting often makes it more efficient to perform operations on remote memory than paying the cost of bringing the objects into local memory. After registering the active component at the remote, data structure engineers invoke `RemDevice`'s client-side bindings to interact with the remote components. They use `construct` and `destruct` to instantiate and destroy remote components. If an object is not present when dereferencing a remote pointer, the runtime invokes the `read_obj` to swap in the missing object. On evacuation, the evacuator invokes `write_obj` to swap out cold objects and `delete_obj` to release dead objects. In addition, the `compute` method invokes a custom function, executing a lightweight computation on the remote server. This is useful, for example, for efficiently aggregating a sum across objects in a data structure without wasting network bandwidth to bring all objects into local memory first.

```
class RemDevice {
  void register_active_component(DSType, ActiveComponent&);
  DSID construct(DSType, ByteArray params);
  void destruct(DSID);
  void read_obj(DSID, ObjID, ByteArray& obj_data);
  void write_obj(DSID, ObjID, ByteArray obj_data);
  bool delete_obj(DSID, ObjID);
  void compute(DSID, OpCode, ByteArray in, ByteArray& out);
};
```

Listing 4: AIFM remoteable device API.

We implemented remote active components to improve the performance of hashtables (§8.2.1) and DataFrames (§8.1.2).

4.2.5 Semantic Hints

AIFM's APIs allow injecting information about application- and object-specific semantics into the runtime.

Hotness tracking. To dereference a remoteable pointer, the user invokes our library, which sets the hot bit of the pointer. Under memory pressure, the memory evacuator uses this hotness information to ensure that frequently accessed objects are local. On evacuation, the evacuators clear the hot bit. AIFM initialization allows developers to customize the number of hot bits to use in the pointer (up to eight) and the replacement policy by data structure ID. With several hot bits, AIFM supports, *e.g.*, a CLOCK replacement policy [72].

Prefetching. AIFM includes a library that data structures can use to maintain a per-thread window of the history of dereferenced locations and predict future accesses using a finite-state machine (FSM). It updates the window and the FSM on each dereference. The FSM detects patterns of sequential access and strided access. When a pattern is detected, it starts prefetcher threads that swap in objects from the remote server. With enough prefetching, application threads always access local memory when dereferencing remoteable pointers. The library estimates the prefetch window size conservatively using the network bandwidth-delay product. Data structure engineers can also add custom prefetching policies.

Nontemporal Access[1]. For remoteable pointers to objects without temporal locality, it makes sense to limit the local memory used to store their object data. This avoids polluting local memory, which multiple data structures may share, with data that a data structure engineer knows is unlikely to be accessed again. To achieve this, AIFM's pointer API supports *non-temporal* dereferences (Listing 5). This immediately marks the object pointed to by `rmt_ptr` as reclaimable, though the actual evacuation happens only after the

[1] We use "nontemporal" in the sense of x86's nontemporal load/store instructions [35], which conceptually bypass the CPU cache to avoid pollution.

```
DerefScope scope;
// non-temporal dereference ⇓ allows immediate reclaim
T* p1 = rmt_ptr1.deref_mut<true>(scope);
// temporal deref; deref_mut(scope) works too
T* p2 = rmt_ptr2.deref_mut<false>(scope);
```

Listing 5: Non-temporal and temporal dereferences.

`DerefScope` ends. Without a hint, a dereference is temporal by default; §8.1.1 evaluates the benefit of the hint.

5 AIFM Runtime

AIFM's runtime is built on "green" threads (light-weight, user-level threading), a kernel-bypass TCP/IP networking stack, and a pauseless memory evacuator. Applications link the runtime into their user-space process. This allows us to co-design the runtime with AIFM's abstractions and provides high-performance far memory without relying on any OS kernel abstractions.

Two high-level objectives guide our runtime design: (*i*) the runtime should productively use the cycles spent waiting during the inevitable latency when fetching objects from remote memory; and (*ii*) application threads should never have to wait for the memory evacuator.

5.1 Hiding Remote Access Latency

We want to hide the latency of fetching data from far memory by doing useful work during the fetch.

Existing OS kernel threads pay high context-switching costs: *e.g.*, on Linux, rescheduling a task takes around 500ns. These costs are a nontrivial fraction of remote memory latency, so Linux and Fastswap adopt a design where they busy-spin while waiting for a network response [6]. This avoids context-switch overheads, but also wastes several microseconds of processing time. This approach also places tremendous pressure on network providers to support even lower latency to reduce the amount of wasted cycles [9, 28]. AIFM takes a different approach: it relies on low-overhead green threads to do application work while waiting for remote data fetches.

Consistent with literature on garbage collection (GC), we refer to normal application threads as *mutator threads* in the following. Each mutator thread accesses far memory, blocking whenever it needs to fetch a remote object. When that happens, another mutator thread can run and make productive use of available CPU cycles. Moreover, AIFM's runtime spawns prefetcher threads to pull in objects that it predicts will be dereferenced in the future, allowing it to avoid blocking mutator threads when the predictions are correct.

Using green threads, AIFM tolerates network latency without sacrificing application-level throughput, wasting fewer cycles than systems that busy-poll for network completion.

5.2 Remoteable Memory Layout

For the local memory managed by AIFM, its runtime embraces the idea of log-structured memory [63], which splits

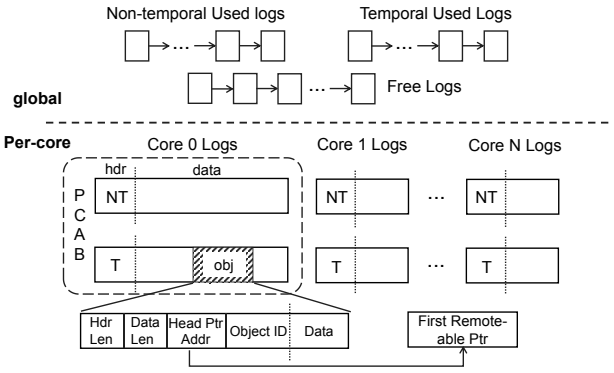


Figure 5: The layout of local remoteable memory in AIFM. There are three global lists: a free list, a temporal used list, and a non-temporal used list. Each list stores many logs, and each log stores many objects. There is a per-core allocation buffer (PCAB) that keeps two free logs to allocate new objects, one log for temporal objects, the other for non-temporal ones.

and manages the local remoteable memory in the granularity of logs (Figure 5). The log size is 2MB, which helps reduce TLB misses by allocating huge pages. The runtime maintains three global lists: a free list, a non-temporal used list, and a temporal used list. Each list stores many logs. For core scalability, each core owns two logs for new allocations: one log for temporal objects, the other for non-temporal ones. The logs are kept in a per-core allocation buffer (PCAB). To allocate an object, the runtime first tries to allocate from a log in the PCAB. If that log runs out of space, the runtime appends the log to the global non-temporal or temporal used list, and obtains a new log from the global free list. To free an object, the runtime marks the object as free. AIFM leverages a mark-compact evacuator to achieve a low memory fragmentation ratio, as shown with other copying log allocators [63].

A log has a 1B header indicating whether it stores non-temporal or temporal data. The remaining space stores objects. Each object has a `Hdr Len` bytes header and a `Data Len` bytes data. The 6-byte `Head Ptr Addr` stores the address of the remoteable pointer that points to the object. For a unique pointer, `Head Ptr Addr` stores the address of the only pointer; for a shared pointer, it stores the address of the first shared pointer in the chain. Dead objects have `Head Ptr Addr` set to `nullptr`. The variable-sized `Object ID` stores the object's unique identifier. The header is used on evacuation, when the runtime passes the object ID to write/delete endpoints on the remote device and the remoteable pointer address to the evacuation handler, and when the runtime swaps in an object and passes the object ID to the remote device.

5.3 Pauseless Memory Evacuator

Upon memory pressure, the runtime's memory evacuator moves cold objects to the remote server. Like with many garbage collectors in managed languages, a key feature of AIFM is to allow mutator threads to run concurrently while

the runtime evacuates local memory. The evacuator executes four phases in sequence, described in the following paragraphs. To ensure correctness under race conditions, the evacuator maintains an invariant: *it only starts to move object O after setting the mutator-side synchronization barrier on accessing O*. The evacuator sets the barrier by setting the pointer evacuation bit (phase 2). The RCU writer wait (phase 3) ensures all mutators have observed the set bits to enforce the timing order in the invariant.

1. Log Selection Phase. The goal of the evacuator is to maintain the local free memory ratio above the *min_free_ratio* (0.12 by default). The *master thread* of the evacuator picks $total\_log\_cnts \cdot (current\_free\_ratio - min\_free\_ratio)$ of logs to be evacuated. The evacuator picks logs in FIFO order from the global non-temporal used list, and then picks from the global temporal used list if necessary, to prioritize non-temporal objects. AIFM could also use more sophisticated schemes, *e.g.*, prioritizing logs by occupancy and age [23].

2. Concurrent Marking Phase. The master evacuation thread spawns *worker threads* and divides the previously-selected logs among them. Each worker thread iterates through the objects in its logs to find live objects. For each such object, the worker sets the evacuation bit of all remoteable pointers of the object by traversing the pointer chain starting from the head pointer address (i.e., the `Head Ptr Addr` field). This marks the object for evacuation.

3. Evacuator Waiting Phase. The runtime can evacuate objects only when they are not being dereferenced by mutator threads. Rather than following a naive approach of having mutators and the evacuator to acquire a per-object lock—which would impose high overhead on the hot path of mutators accessing local objects—AIFM uses an approach inspired by read-copy-update (RCU) synchronization. AIFM's runtime treats mutators as RCU readers and the evacuator master thread as an RCU writer, thereby moving the synchronization overhead to the evacuator. This choice makes sense because (*i*) the mutators do application work, so AIFM should steer overhead away from them; and (*ii*) evacuation is a rare event. The result is that the evacuator master thread waits for a quiescent period to ensure all mutator threads have witnessed the newly-set evacuation bits.

If a mutator thread subsequently dereferences a pointer to an object that the runtime is evacuating, the mutator sees that the evacuation bit is set. A naive approach would now block the mutator thread while the evacuation bit is set. Instead, AIFM opts for an approach that avoids such pauses: the mutator copies the object to another log in its PCAB, and then executes a compare-and-swap (CAS) on the head remoteable pointer (which serves as a synchronization point) to simultaneously clear the evacuation bit, set the present bit, and set the new data location. This CAS will race with the evacuator (see next phase below). If the CAS succeeds, the mutator copied an intact object, so it obtains a local reference. The mutator then updates all pointers in the pointer chain with the head pointer metadata and continues executing. If the CAS fails, the evacuator has already changed the remoteable pointer to remote status, so the mutator's copy of the object may be corrupt. Consequently, the mutator frees the copy it made and obtains a remote reference.

4. Concurrent Evacuation Phase. The master thread spawns more worker threads to evacuate objects and run their evacuation handlers. Again, the master divides the previously selected logs among the workers. Each worker iterates through each log and each object within the log. For each cold object, the worker copies the object to the remote and executes a CAS on the head remoteable pointer to simultaneously clear the presence bit and set the remote pointer metadata. If the CAS succeeds, the object has been evacuated, and the worker updates all pointers in the pointer chain with the head pointer metadata and invokes the evacuation handler. Otherwise, a mutator thread succeeded with a racing CAS and has copied the object to another location. Either way, the log entry is now unused and reclaimable. For each hot object, the worker compacts and copies it into a new log, updates the object address in the remoteable pointers, and resets the hot bits.

5.4 Co-design with the Thread Scheduler

Evacuation is an urgent task when the runtime is under memory pressure. With a naive thread scheduler, evacuation can be starved by mutator threads, leading to out-of-memory errors and application crashes. There are two challenges that we need to address. First, a large number of mutator threads may allocate memory faster than evacuation can free memory. Second, evacuation sometimes blocks on mutator threads in a dereference scope, and this creates a dilemma. On one hand, the scheduler needs to execute mutator threads so they can unblock evacuation. On the other hand, executing mutator threads may consume more memory.

To address these issues, we co-design the runtime's green thread scheduler with AIFM to prioritize the activities necessary for evacuation, both in mutator threads and evacuation threads. First, each thread keeps a status field that is set by the AIFM runtime and read by the scheduler, which allows the scheduler to know whether a thread is in a dereference scope. The scheduler runs a multi-queue algorithm and assigns the first priority to mutators in a dereference scope, second priority to evacuation threads, and third priority to other mutator threads. Second, to avoid priority inversion [42] when the system is short of memory, the allocation function in the AIFM runtime triggers a signal to all running threads to force them to yield their cores back to the scheduler for re-scheduling.

6 Remoteable Data Structure Examples

We implemented six remoteable AIFM data structures.

Array. The remoteable array consists of a native array of `RemUniquePtr`s. Each pointer points to an array element to enable fine-grained data placement decisions. Alternatively, users can configure the pointed object as multiple consecutive

array elements to reduce the memory overhead of pointer metadata. The object IDs of pointers are their remote-side object addresses. The prefetcher records accessed indices at all array access APIs; it starts prefetching when detecting a strided access pattern.

Vector. The remoteable vector is similar to the remoteable array except that it is dynamically sized, and uses a `std::vector` to store `RemUniquePtrs`. Additionally, the vector has an active remote component that supports offloading operations like copies and aggregations, which are used by the DataFrame application (§8.1.2).

List. The remoteable list is similar to the remoteable vector, except that it uses a local list that stores `RemUniquePtrs` to support efficient `insert` and `erase` operations. The list supports traversals in forward and reverse directions, which offers strong semantic hints to the prefetcher. When detecting a direction, the prefetcher walks through the local list in the same direction to prefetch remote list objects.

Stack and Queue. The remoteable stack and queue are simple wrappers around remoteable lists.

Hashtable. The remoteable hashtable consists of a table index (stored on the local heap) and key-value data (stored in AIFM's remoteable heap). In the index, each hash bucket stores a `RemUniquePtr` to a key-value object. The object IDs of pointers are their hashtable keys. The hashtable has an active remote component that maintains a separate hashtable in remote memory. In this architecture, the local hashtable is a cache (inclusive or exclusive) of its remote counterpart. When the referenced object is missing from the local cache, the active remote component assists the chain lookup at the remote hashtable to avoid multiple network round-trips. Data structure engineers might also realize different hashtable designs via AIFM's APIs.

7 Implementation

AIFM's implementation consists of the core runtime library (§5) and the data structure library (§6). The core runtime is built on top of Shenango [55] to leverage its fast user-level threading runtime and I/O stack. AIFM is written in C and C++, with 6,451 lines in the core runtime, 5,535 lines in the data structure library, and 750 lines of modifications to the Shenango runtime. The system runs on unmodified Linux.

We integrated two far memory backends into AIFM: a remote memory server based on a DPDK-based TCP stack, and an NVMe SSD using an SPDK-based storage stack. Unlike the remote memory backend, the SSD backend does not support active remote components (since the storage drive does not have a general compute unit), and it has an inherent I/O amplification because it is limited to a fixed block size. Our evaluation focuses on the remote memory backend.

The current implementation has some limitations. First, we do not support TCP offloading or RDMA, which would reduce CPU overhead of our runtime. Second, a local compute server connects to a single remote memory server, and the remote memory cannot be shared by different clients. Finally, the local and remote memory size cannot be changed at runtime. We plan to address them in the future.

8 Evaluation

Our evaluation of AIFM seeks to answer three questions:

1. What performance does AIFM achieve for end-to-end applications, including ones that combine multiple remoteable data structures? (§8.1)
2. How does AIFM's performance compare to a state-of-the-art far memory system, Fastswap [6]? (§8.1–§8.2)
3. What factors contribute to AIFM's performance? (§8.3)

Setup. We run experiments on two `xl170` nodes on CloudLab [25] with 10-core Intel Xeon E5-2640 v4 CPUs (2.40 GHz), 64GB RAM, and a 25 Gbits/s Mellanox ConnectX-4 Lx MT27710 NIC. We enabled hyper-threads, but disabled CPU C-states, dynamic CPU frequency scaling, transparent huge pages, and kernel mitigations for speculation attacks in line with prior work [55]. We use Ubuntu 18.04.3 (kernel v5.0.0) and DPDK 18.11.0, except for experiments with Fastswap, which use Linux kernel v4.11, the latest version Fastswap supports. All AIFM experiments use the default configuration settings and the default built-in prefetchers of remoteable data structures. We do not tune prefetching policy specifically for evaluated applications.

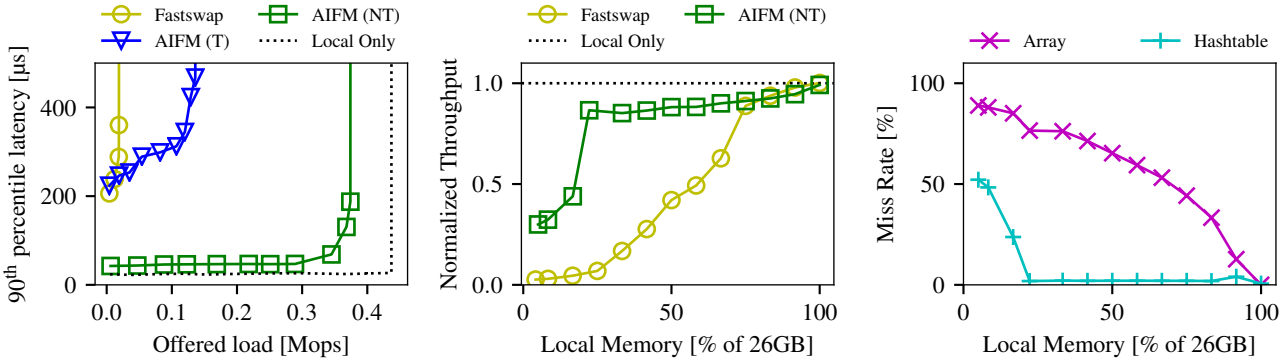
8.1 End-to-end Performance

We evaluate AIFM's end-to-end performance with two applications. First, we designed a synthetic application that mimics a typical web service frontend to understand AIFM's performance with multiple remoteable data structures and the impact of semantic hints. Second, we also ported an open-source C++ DataFrame library [16] with an interface similar to Pandas [56] to AIFM, and use it to understand the porting effort required and AIFM's performance for an existing application.

8.1.1 Synthetic Web Service Frontend

In response to client requests, the application fetches structured data (*e.g.*, a list of user IDs) from an in-memory key-value store, and then uses the retrieved values to compute an index into a large collection of 8KB objects (*e.g.*, profile pictures). Finally, the application fetches one 8KB object, encrypts it, and compresses it for the response to the client.

This application uses our remoteable hashtable (for the key-value pairs) and our remoteable array (for the 8KB objects). Each client request looks up 32 keys in the hashtable and fetches a single 8KB array element. We load the hashtable with 128M key-value pairs (10GB total data, of which 6GB are index data and 4GB are value data), and create an array of 2M objects of 8KB each (16GB total). The two data structures share 5GB of available local memory, *i.e.*, the local memory size is 19% of the total data set size. We generate closed-loop client requests from a Zipf distribution with parameter *s*: a uni-



(a) Latency as a function of offered load. **(b)** Throughput as a function of local memory. **(c)** AIFM miss rates in Figure 6b.

Figure 6: In a web frontend-like application with a hashtable and array, AIFM outperforms Fastswap by 20× (a) and achieves 90% of local memory performance with 5× less memory (b), as non-temporal array access avoids polluting local memory (c). "AIFM(NT)": non-temporal access; "AIFM(T)": temporal access; "Local Only": entire working set in local memory.

form distribution corresponds to $s = 0$, while values of s close to 1 indicate high skew. Each request accesses Zipf-distributed keys in the hashtable and uses their values to calculate an (also Zipf-distributed) array index to access; the request then encrypts the array data via AES-CBC using crypto++ [22] and compresses the result using Snappy [30]. We compare two AIFM settings—with and without non-temporal dereferences for array elements—against Fastswap [6] and an idealized baseline with all 26GB in local memory. A good result for AIFM would show improved performance over Fastswap, a benefit to non-temporal array accesses, and performance not much lower than keeping the entire data in local memory.

Figure 6a shows a throughput-latency plot for a Zipf parameter of $s = 0.8$ (*i.e.*, a skewed distribution). The x-axis shows the offered load in the system, and the y-axis plots the measured 90$^{\text{th}}$ percentile latency. Each setup eventually encounters a "hockey-stick" when it can no longer keep up with the offered load. Fastswap tolerates a load of up to 19k requests/second, but its overheads and the amplification for the hashtable lookups quickly dominate. AIFM with a temporal array dereference scales 7× further, but fails to keep up beyond 140k requests/second because the 8KB array accesses pollute its local memory. To make room for an 8KB array element, the runtime often evicts hundreds of hashtable entries, causing a high miss rate on hashtable lookups. AIFM with non-temporal access to the array, however, scales to 370k requests/second (20× Fastswap's maximum throughput). This is 16% lower throughput than the 440k requests/second achieved by an idealized setup with 26GB in local memory. In other words, AIFM achieves 84% of the performance of an entirely local setup with 5× less local memory.

Additional local memory helps bring AIFM performance closer to the in-memory ideal. Figure 6b shows the percentage of the all-local memory throughput achieved by the non-temporal version of AIFM when varying the local memory size (on the x-axis, as a fraction of 26GB). While Fastswap's throughput starts near zero and grows roughly in proportion to the local memory size, AIFM's throughput starts at 30% of the ideal and quickly reaches 85% of the in-memory throughput at 5.0GB local memory (20% of 26GB).

Figure 6c illustrates why this happens. At the left-hand side of the plot (5% local memory), AIFM sees high miss rates in both hashtable (52%) and array (89%). But as local memory grows, the hashtable miss rate quickly drops to near-zero, since AIFM's non-temporal dereferences for the array ensure that most of the local memory is dedicated to hash table entries. Correspondingly, the array miss rate drops more slowly and in proportion to the local memory available. By contrast, Fastswap (not shown here) has high miss rates in both data structures, as its page-granular approach manages local memory inefficiently.

8.1.2 DataFrame Application

The DataFrame abstraction, popularized in Pandas [56], provides a convenient set of APIs for data science and ML workloads. A DataFrame is a table-structured, in-memory datastructure exposing various slicing, filtering, and aggregation operations. DataFrames often have hundreds of columns and millions of rows, and their full materialization in memory often pushes the limits of available memory on a machine [54, 57, 60]. By making remote memory available, AIFM can help data scientists interactively explore DataFrames without worrying about running out of memory.

We ported a popular open-source C++ DataFrame library [16] to AIFM's APIs. The primary data structure used in the library is an `std::vector` storing DataFrame columns and indexes, and we replaced this vector with the AIFM-enabled equivalent. In addition, we also added support for offloading key operations with low compute intensity but high memory access frequency to the remote side. We achieve this by offloading three operations using AIFM's remote device

		Offloaded Rem. Dev. Operations		
		Copy	Shuffle	Aggregate
DataFrame API	Filter	✓		
	Range extraction	✓		
	Add column/index	✓		
	Sort by column		✓	
	GroupBy		✓	✓

Table 1: DataFrame APIs (rows) and the offloaded operations they use via AIFM's remote device API (columns). Copy and Shuffle are memory-only operations, while Aggregate performs light remote-side computation.

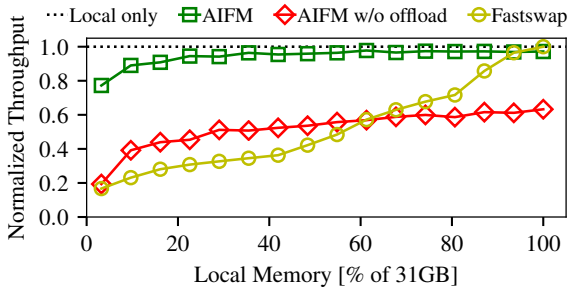


Figure 7: An AIFM-enabled DataFrame library [16] achieves 78–97% of in-memory throughput for a data analytics workload [53], outperforming Fastswap. Offloading operations with low compute intensity is crucial to AIFM's performance.

API (§4.2.4). The *Copy* and *Shuffle* operations copy a vector (*i.e.*, a DataFrame column), with shuffle also reordering rows by index positions in another column; *Aggregate* computes aggregate values (sums, averages, etc.). These three operations are used in five DataFrame API calls, including filters, column creation, sorts, and aggregations (Table 1). To achieve coverage sufficient to run the New York City taxi trip analysis workload [53], we modified 1,192 lines of code in the DataFrame library (which has 24.3k lines), and wrote 233 lines of remote device code. These modifications took one author about five days.

We benchmark our AIFM-enabled DataFrame with the Kaggle NYC taxi trip analysis workload [53], which explores trip dimensions including the number of passengers, trip durations, and distances, on the NYC taxi trip dataset [74] (16GB). The workload's full in-memory working set is 31GB. In the experiment, we vary the size of available local memory between 1GB and 31GB. We compare AIFM with Fastswap and a baseline with all data in local memory. In addition, we also investigate the impact of offloading on this workload, which consists of an operation with low compute intensity (*Aggregate* in Table 1) and some pure memory-copy operations (*Copy* and *Shuffle*). We would hope to find AIFM outperform Fastswap and come close to the local memory baseline.

Figure 7 shows the results. AIFM achieves 78% of in-memory throughput even with 1GB of local memory (3.2%) and exceeds 95% of ideal performance from about 20% (6GB) local memory. Fastswap, by contrast, achieves only 20% of in-memory performance at 1GB and only comes close to it once

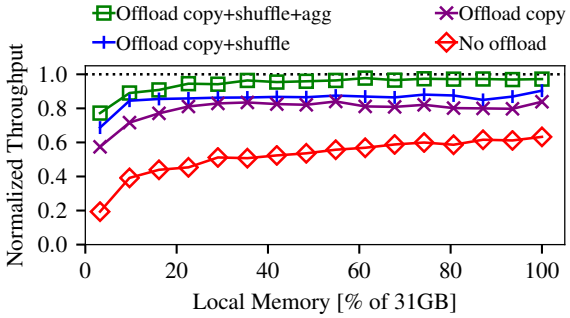


Figure 8: Performance gains from offloading the operations in Table 1. AIFM benefits most from offloading *Copy*, which increases throughput by 18–38%.

over 90% of the working set are in local memory. AIFM's high performance comes from avoiding Fastswap's page fault overheads, and from reducing expensive data movements over network by offloading operations with low compute intensity. Without offloading, AIFM outperforms Fastswap until 60% of the working set are local, as Fastswap incurs frequent minor faults. Beyond 60%, the fault rate in Fastswap drops sufficiently for most memory accesses to outperform AIFM's dereference-time overhead for low compute intensity operations (*e.g.*, memory copies). Offloading these operations to the remote side helps AIFM avoid this cost, while high compute-intensity operations amortize the dereference cost and happen locally. We also prototyped a batched API for AIFM that amortizes the dereference overhead across groups of vector elements when offloading is not possible, and found that it improves AIFM's throughput without offloading to 60–80% of in-memory throughput. We believe this could make a good future addition to AIFM's API to speed up low compute intensity operations if they must be performed locally.

Figure 8 breaks down the effect of offloading. Offloading *Copy* contributes the largest throughput gains (18%–38%); offloading *shuffle* contributes 2.9%–13%; and offloading *Aggregate* contributes 4.5%–12%. These results show that AIFM achieves high performance with small local memory for a real-world workload, and that AIFM's operation offloading is crucial to good performance when a workload includes operations with low compute intensity.

8.2 Data Structures

We pick two representative data structures—the hashtable and the array—from §6. We evaluate them in isolation, and explore the impact of prefetching, non-temporal local storage, and read/write amplification-reducing techniques.

8.2.1 Hashtable

Hash tables provide unordered maps that typically see random accesses, often with high temporal locality. A remoteable hash table should benefit from temporal caching of popular key-value (KV) pairs in local memory. Note that with AIFM, the caching policy is controlled by the data structure engi-

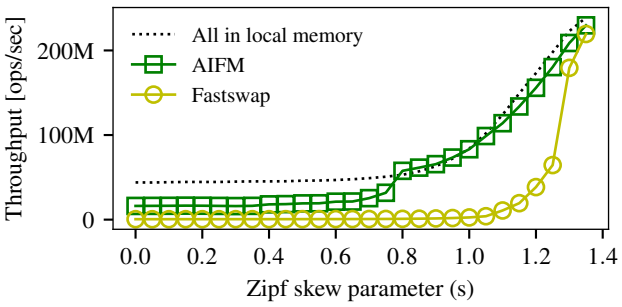
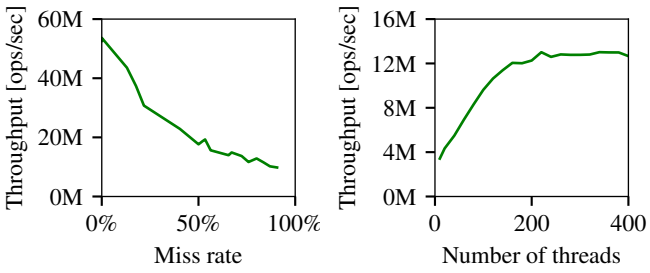


Figure 9: An AIFM hash table is competitive with local memory when the access distribution is skewed (Zipf factors ≥ 0.8), and outperforms a hashtable in Fastswap by up to 61× as Fastswap suffers from amplification and other overheads.

neer, while with Fastswap (or any swap-based far memory system), the caching policy is determined by the kernel page-reclamation policy, which in turn is based on page-granular hotness information.

Comparison. We evaluate the hashtable over Fastswap and AIFM with a memcached-style workload that issues `GET` requests, with keys sampled from a Zipf distribution whose parameter s we vary. Our key and value sizes are based on those reported for Facebook's USR memcached pool [8]. We load the hash table with 128M KV pairs (10GB total data), and compare performance to a baseline that keeps the entire hash table in local memory. Fastswap and AIFM instead allow a maximum of 5GB local data, split as follows. In Fastswap, the OS manages the both hashtable index (6GB) and value data (4GB) in swapable memory, with least recently used (LRU) [77] eviction at page granularity to decide on remote pages. In AIFM, we provision 3GB local memory region for index data and the other 2GB local memory region for value data; the runtime manages them separately. The hashtable's own object-granular CLOCK replacement algorithm guides AIFM's memory evacuator to pick KV pairs to evict to remote memory. In this experiment, we use a hashtable configured as an exclusive cache, *i.e.*, the evacuation handler removes local index entries for remote key-value pairs.

Figure 9 shows the throughput achieved as a function of the Zipf parameter s, ranging from near-uniform at zero to highly skewed at $s = 1.35$. AIFM achieves about 17M operations/second at low skew ($\approx 60\%$ miss rate at $s = 0$), about one third of the 53M operations/second that a fully-local hash table achieves. As skew increases and the miss rate drops, AIFM comes closer to local-only performance: for example, at $s = 0.8$ (1% miss rate), it reaches 57M operations/second; and from $s = 0.8$, it matches the performance of the local-only hashtable. Fastswap, by contrast, sees a throughput of 0.54M operations/second at $s = 0$ (30× less than AIFM) and only matches the local-only baseline beyond $s = 1.3$. At $s = 0.8$, AIFM has its largest advantage over Fastswap (61×).



(a) `GET` throughput as a function of the miss rate. **(b)** `GET` throughput as a function of thread count (80% miss rate).

Figure 10: AIFM hash table microbenchmarks.

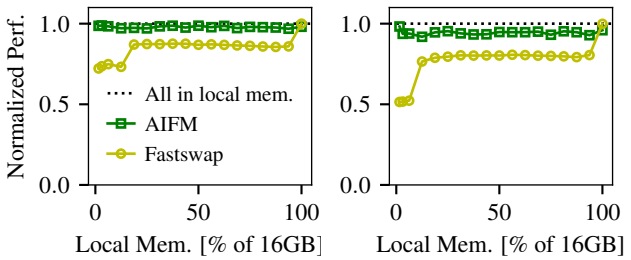
This difference comes from three factors against Fastswap: (*i*) amplification due to page-granular swapping, (*ii*) lack of per-KV pair hotness information, and (*iii*) the overheads of kernel paging. Since a page contains 128 key-value pairs, page-granular swapping incurs up to 128× read and write amplification. This amplification increases the network bandwidth required and pollutes the local memory, increasing Fastswap's miss rate with identical memory available. For example, at $s = 1.25$, Fastswap still uses 140MB/s of network bandwidth, while AIFM's bandwidth use rapidly drops beyond $s = 0.8$. Fastswap also cannot swap out only cold key-value pairs, as a page contains entries with varying hotness, but the kernel tracks access only at page granularity. Finally, Fastswap incurs the cost of kernel crossings, page faults, identifying and reclaiming victim pages (38% of cycles at $s = 0.8$) and wasted cycles waiting for I/O (49%). AIFM's overheads are limited to running the evacuator (0.8% of cycles at $s = 0.8$), TCP stack overheads (1.7%), and thread scheduler overhead (14%).

Microbenchmarks. Figure 10a shows how hash table performs at different miss rates when requests are uniformly, rather than Zipf-distributed. It achieves a best-case throughput of 53M requests/second, reduced to 10M requests/second when it is close to 100% miss rate. Figure 10b measures, for the same uniform distribution and an 80% miss rate, the throughput AIFM achieves with an increasing number of application threads. Up to 160 threads, AIFM extracts more throughput by scheduling additional requests while it waits for requests to complete.

8.2.2 Array

Depending on the access pattern, an array may benefit from caching (for random access with temporal locality), prefetching (for sequential access), and non-temporal storage (if there is no temporal locality).

We evaluate our array with the Snappy library [30]. The benchmark performs in-memory compression/decompression by reading input files from a `RemArray` and writing output files to another `RemArray`. For benchmarking compression, we use 16 input files of 1GB each. For decompression, we use



(a) Compression. **(b)** Decompression.

Figure 11: AIFM achieves nearly identical performance to local memory when compressing/decompressing an array with Snappy [30] (sequential access), and outperforms Fastswap.

30 input files of 0.5GB each. The compression ratio is around 2. Both operations perform streaming, sequential access to the array and never revisit any object. We compare Fastswap and an ideal, completely local in-memory baseline.

AIFM's array prefetcher captures application semantics through the array access APIs and performs prefetching entirely in user space. OS-based paging systems, by contrast, must rely on page faults (major faults for unprefetched pages and minor faults for prefetched pages) to pass application semantics, which imposes high overheads. For each system, we measure performance with different amounts of local memory available (for Fastswap, we restrict memory via cgroups; for AIFM, we set the local memory size). A good result would avoid AIFM's slow path, as every far pointer dereference would find local data already.

Figure 11 shows the results. We see that AIFM achieves performance close to the in-memory baseline, independent of the local memory size, while Fastswap's performance depends on local memory size and only matches AIFM when nearly all memory is local. This demonstrates the benefit of AIFM's non-temporal access and prefetching.

8.3 Design Drill-Down

We now evaluate specific aspects of the AIFM design using microbenchmarks.

8.3.1 Fast/Slow Path Costs

AIFM seeks to provide access to local objects with latency close to normal memory access. This means that AIFM's remoteable pointer must minimize overheads on the "fast path", when no remote memory access is required.

We measured the hot path latency of dereferencing a `RemUniquePtr` and compared it to the latency for dereferencing a C++ `unique_ptr`, both when the pointer and data pointed to are cached and uncached. Figure 12a shows that AIFM offers comparable latency to an ordinary C++ smart pointer. For an object in L1 cache, AIFM has a 4× latency overhead: four micro-ops *vs.* a single pointer dereference operation. In practice, modern CPU's instruction-level parallelism

90th percentile latency [cycles]	*read*	*write*
C++ `unique_ptr` (uncached)	570	408
AIFM object (uncached)	489	309

(a) Hot path (local object).

90th percentile latency [cycles]		*read*	*write*
Fastswap total		23,712	26,382
... of which	RDMA transfer (4KB)	16,521	16,521
	Overheads	7,191	9,861
AIFM total (64B object)		18,582	18,369
... of which	TCP transfer (64B)	17,694	17,673
	Overheads	888	696
AIFM total (4KB object)		27,183	27,279
... of which	TCP transfer (4KB)	26,055	26,121
	Overheads	1,128	1,158

(b) Cold path (remote object).

Figure 12: AIFM is competitive with an ordinary pointer dereference, and it has lower overheads than Fastswap.

hides some of this latency, and we observe a 2× throughput overhead for L1 hits.

We also measured AIFM's cold path latency, and compared it to Fastswap's. Fastswap always fetches at least 4KB from the remote server, but its RDMA backend is faster than AIFM's TCP backend. This might amortize some of the overheads associated with page-granular far memory that Fastswap suffers from. A good result would show AIFM with comparable latency to Fastswap for large objects (4KB), and lower latency for small objects (64B).

Figure 12b shows the results. While Fastswap's raw data transfers are indeed faster than AIFM's, AIFM achieves lower latency for cache-line-sized (64B) objects due to its 10× lower overheads. For 4KB objects, AIFM is close to Fastswap, but has 10% higher latency on reads; AIFM with an RDMA backend would come closer. In addition, AIFM can productively use its wait cycles, which yields a 1.8–6.8× throughput increase over Fastswap (Figure 1).

8.3.2 Operating Point

AIFM is designed for applications that perform some compute for each remoteable data structure access, as this compute allows AIFM to hide the latency of far memory by prefetching. But if an application has a huge amount of compute per data structure access, AIFM will offer limited benefit over page-granular approaches like Fastswap, despite their overheads. We ran a sensitivity analysis with a synthetic application that spins for a configurable amount of time in between sequential accesses into a remoteable array. This should allow AIFM's prefetcher to run ahead and load successive elements before they are dereferenced. We compare to Fastswap, which we configure with the maximum prefetching window (32 pages).

Figure 13 shows the results, normalized to the benchmark runtime against a purely in-memory array. AIFM becomes competitive with local memory access from about 1.2µs of compute between array accesses. Fastswap's overheads amor-

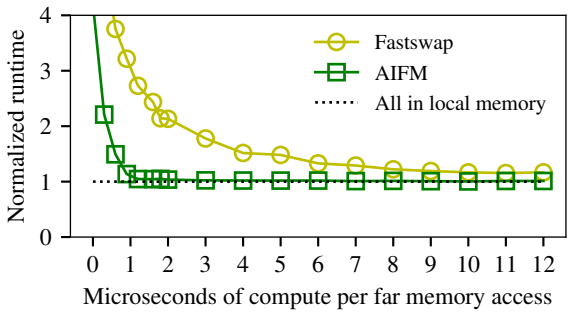


Figure 13: AIFM becomes competitive with local memory access at around 1.2µs of compute per sequential far memory access (4KB object) in a microbenchmark, while kernel-based swapping mechanisms require higher compute ratios (ca. 50µs per memory access; not shown) to compete.

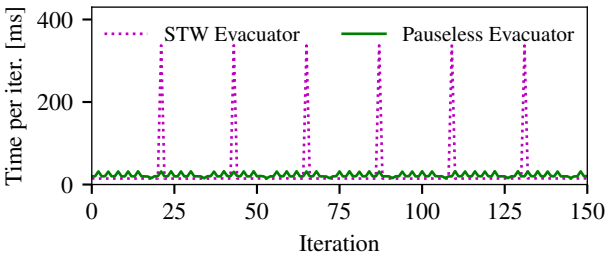


Figure 14: Pauseless evacuation is essential for low latency accesses: a stop-the-world (STW) evacuator frequently encounters 10× higher latency as it swaps out objects.

tize more slowly—its line converges with AIFM's around 50µs of compute per array access. This demonstrates that AIFM supports efficient remote memory in a wider range of applications than page-granular approaches like Fastswap.

8.3.3 Memory Evacuator

We evaluate two key aspects of AIFM's memory evacuator design: the choice to never pause mutator threads (§5.3) and the thread scheduler co-design (§5.4).

Pauseless Evacuation. In this experiment, we run 10 mutator threads (the number of physical CPU cores in our machine) that keep entering the dereference scope, dereferencing and marking dirty 4MB of data each time. Therefore, the runtime periodically triggers memory evacuation. We compare AIFM's pauseless evacuator design to a stop-the-world memory evacuator, and measure the latency per mutator iteration (4MB write). Figure 14 shows that a stop-the-world evacuator design causes periodic mutator latency spikes up to 340ms. By contrast, AIFM's pauseless evacuator consistently runs an iteration in about 25ms. (The tiny spikes of the pauseless line are mainly caused by hyperthread and cache contention between evacuators and mutators.) This confirms that a pauseless evacuator is essential to consistent application performance.

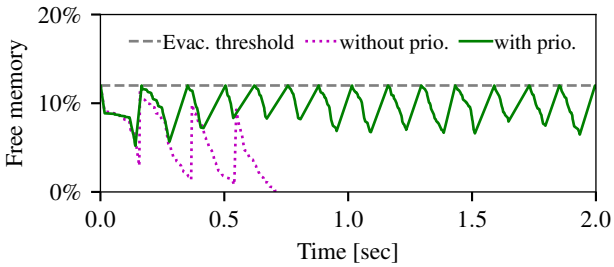


Figure 15: Thread prioritization in the runtime is essential to ensure that evacuation always succeeds. 12% free memory is the threshold for AIFM to trigger evacuation.

Thread Scheduler Co-design. In this experiment, we run 100 mutator threads that each iterates to read 1MB of data from a remoteable array and perform 20ms of computation. We run AIFM with the scheduler's thread prioritization (§5.4) enabled and disabled, and measure the free local memory over time. For a responsive system, local memory should never run out entirely, and the evacuator should be able to free memory fast enough to keep up with the mutators.

Figure 15 shows that the runtime without prioritization fails to keep up and runs out of memory after around 0.7 seconds. AIFM's prioritizing scheduler, on the other hand, ensures that sufficient memory remains available. This illustrates that the benefit of co-locating thread scheduler and memory evacuator in a user-space runtime.

9 Conclusion

We presented Application-Integrated Far Memory (AIFM), a new approach to extending a server's available RAM with high-performance remote memory. Unlike prior, kernel-based, page-granular approaches, AIFM integrates far memory with application data structures, allowing for fine-grained partial remoting of data structures without amplification or high overheads. AIFM is based on four key components: (*i*) the remote pointer abstraction; (*ii*) the pauseless memory evacuator; (*iii*) the data structure APIs with rich semantics; (*iv*) and the remote device abstraction. All parts work together to deliver high performance and convenient APIs for application developers and data structure engineers.

Our experiments show that AIFM delivers performance close to, or on par with, local DRAM at operating points that prior far memory systems could not efficiently support.

AIFM is available as open-source software at https://github.com/aifm-sys/aifm.

Acknowledgements

We thank our shepherd Emmett Witchel, the anonymous reviewers, and members of the MIT PDOS group for their helpful feedback. We appreciate Cloudlab [25] for providing the experiment platform used. This work was supported in part by a Facebook Research Award and a Google Faculty Award. Zhenyuan Ruan was supported by an MIT Robert J. Shillman Fund Fellowship.

References

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. "FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. "Remote regions: a simple abstraction for remote memory". In: *USENIX Annual Technical Conference (ATC)*. 2018.

[3] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. "Remote Memory in the Age of Fast Networks". In: *ACM Symposium on Cloud Computing (SoCC)*. 2017.

[4] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. "Designing far memory data structures: Think outside the box". In: *Workshop on Hot Topics in Operating Systems (HotOS)*. 2019.

[5] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. "Write-Rationing Garbage Collection for Hybrid Memories". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2018.

[6] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. "Can Far Memory Improve Job Throughput?" In: *European Conference on Computer Systems (EuroSys)*. 2020.

[7] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. "Treadmarks: Shared memory computing on networks of workstations". In: *Computer* 29.2 (1996).

[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. "Workload Analysis of a Large-Scale Key-Value Store". In: *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 2012.

[9] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. "Attack of the killer microseconds". In: *Communications of the ACM* 60.4 (2017).

[10] John K. Bennett, John B. Carter, and Willy Zwaenepoel. "Munin: Distributed Shared Memory Based on Type-specific Memory Coherence". In: *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 1990.

[11] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. "The End of Slow Networks: It's Time for a Redesign". In: *Proceedings of the VLDB Endowment* 9.7 (2016).

[12] Andrew D. Birrell and Bruce Jay Nelson. "Implementing Remote Procedure Calls". In: *ACM Transactions on Computer Systems (TOCS)* 2.1 (1984).

[13] Hans-Juergen Boehm and Mark Weiser. "Garbage collection in an uncooperative environment". In: *Software: Practice and Experience* 18.9 (1988).

[14] Michael D. Bond and Kathryn S. McKinley. "Tolerating Memory Leaks". In: *ACM SIGPLAN Notices* 43.10 (2008).

[15] Benjamin Brock, Aydın Buluç, and Katherine Yelick. "BCL: A cross-platform distributed container library". In: *International Conference on Parallel Processing (ICPP)*. 2019.

[16] "C++ DataFrame for statistical, Financial, and ML analysis". In: `https://github.com/hosseinmoein/DataFrame` (last accessed on 10/15/2020).

[17] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. "Project PBerry: FPGA Acceleration for Remote Memory". In: *Workshop on Hot Topics in Operating Systems (HotOS)*. 2019.

[18] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. "A cloud-scale acceleration architecture". In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016.

[19] Youmin Chen, Youyou Lu, and Jiwu Shu. "Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing". In: *European Conference on Computer Systems (EuroSys)*. 2019.

[20] Cliff Click, Gil Tene, and Michael Wolf. "The pauseless GC algorithm". In: *ACM/USENIX international conference on Virtual execution environments (VEE)*. 2005.

[21] Douglas Comer and Jim Griffioen. "A New Design for Distributed Systems: The Remote Memory Model". In: *Summer USENIX Conference*. 1990.

[22] "Crypto++ Library 8.2". In: https://www.cryptopp.com/ (last accessed on 10/15/2020).

[23] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. "Garbage-First Garbage Collection". In: *International Symposium on Memory Management (ISMM)*. 2004.

[24] Fred Douglis. "The compression cache: Using on-line compression to extend physical memory". In: *Winter USENIX Conference*. 1993.

[25] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. "The Design and Operation of CloudLab". In: *USENIX Annual Technical Conference (ATC)*. 2019.

[26] "Facebook Folly RCU Library". In: https://github.com/facebook/folly/blob/master/folly/synchronization/Rcu.h (last accessed on 10/15/2020).

[27] Michail D. Flouris and Evangelos P. Markatos. "The Network RamDisk: Using Remote Memory on Heterogeneous NOWs". In: *Cluster Computing* 2.4 (1999).

[28] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. "Network Requirements for Resource Disaggregation". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[29] "Gen-Z: hardware architecture for disaggregated memory". In: https://genzconsortium.org (last accessed on 10/15/2020).

[30] "Google's fast compressor/decompressor". In: https://github.com/google/snappy (last accessed on 10/15/2020).

[31] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. "Efficient Memory Disaggregation with Infiniswap". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2017.

[32] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. "RDMA over Commodity Ethernet at Scale". In: *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. 2016.

[33] "HPE Powers Up The Machine Architecture". In: https://www.nextplatform.com/2017/01/09/hpe-powers-machine-architecture (last accessed on 10/15/2020).

[34] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. "The Garbage Collection Advantage: Improving Program Locality". In: *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2004.

[35] "Intel 64 and IA-32 Architectures Developer's Manual: Vol. 1". In: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html (last accessed on 10/15/2020).

[36] "Java SE documentation. Chapter 6: The Parallel Collector". In: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html (last accessed on 10/15/2020).

[37] Anuj Kalia, Michael Kaminsky, and David Andersen. "Datacenter RPCs can be General and Fast". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2019.

[38] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Using RDMA Efficiently for Key-value Services". In: *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. 2014.

[39] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Design Guidelines for High Performance RDMA Systems". In: *USENIX Annual Technical Conference (ATC)*. 2016.

[40] Samir Koussih, Anurag Acharya, and Sanjeev Setia. "Dodo: A User-level System for Exploiting Idle Memory in Workstation Clusters". In: *IEEE International Symposium on High Performance Distributed Computing (HPDC)*. 1998.

[41] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. "Software-Defined Far Memory in Warehouse-Scale Computers". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019.

[42] Butler W Lampson and David D Redell. "Experience with processes and monitors in Mesa". In: *Communications of the ACM* 23.2 (1980).

[43] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2017.

[44] Kai Li and Paul Hudak. "Memory Coherence in Shared Virtual Memory Systems". In: 7.4 (1989).

[45] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. "Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device". In: *IEEE International Conference on Cluster Computing (CLUSTER)*. 2005.

[46] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. "Imbalance in the cloud: An analysis on alibaba cluster trace". In: *IEEE International Conference on Big Data (Big Data)*. IEEE. 2017.

[47] "Mellanox Innova-2 Flex Open Programmable SmartNIC". In: https://www.mellanox.com/products/smartnics/innova-2-flex (last accessed on 10/15/2020).

[48] Feeley Michael J, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, and Henry M. Levy. "Implementing Global Memory Management in a Workstation Cluster". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 1995.

[49] Christopher Mitchell, Yifeng Geng, and Jinyang Li. "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store". In: *USENIX Annual Technical Conference (ATC)*. 2013.

[50] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. "Latency-tolerant Software Distributed Shared Memory". In: *USENIX Annual Technical Conference (ATC)*. 2015.

[51] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. "Storm: a fast transactional dataplane for remote data structures". In: *ACM International Conference on Systems and Storage (SYSTOR)*. 2019.

[52] "NVIDIA Mellanox BlueField DPU". In: https://www.mellanox.com/products/bluefield-overview (last accessed on 10/15/2020).

[53] "NYC Taxi Trips - Exploratory Data Analysis". In: https://www.kaggle.com/kartikkannapur/nyc-taxi-trips-exploratory-data-analysis/notebook (last accessed on 10/15/2020).

[54] "Opening a 20GB file for analysis with pandas". In: https://datascience.stackexchange.com/questions/27767/opening-a-20gb-file-for-analysis-with-pandas/ (last accessed on 10/15/2020).

[55] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2019.

[56] "pandas - Python Data Analysis Library". In: https://pandas.pydata.org/ (last accessed on 10/15/2020).

[57] "Pandas: Scaling to large datasets". In: https://pandas.pydata.org/pandas-docs/stable/user_guide/scale.html (last accessed on 10/15/2020).

[58] Nathan Pemberton. "Exploring the disaggregated memory interface design space". In: *Workshop on Resource Disaggregation (WORD)*. 2019.

[59] "Persistent Memory Development Kit". In: https://pmem.io/pmdk/ (last accessed on 10/15/2020).

[60] "Quora: Is anyone successful in using Python Pandas while dealing with millions of rows or more than a billion?" In: https://www.quora.com/Is-anyone-successful-in-using-Python-Pandas-while-dealing-with-millions-of-rows-or-more-than-a-billion-If-not-what-else-did-you-do (last accessed on 10/15/2020).

[61] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. "PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2017.

[62] Andy Rudoff. "Persistent memory programming". In: *Login: The Usenix Magazine* 42 (2017).

[63] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. "Log-structured Memory for DRAM-based Storage". In: *USENIX Conference on File and Storage Technologies (FAST)*. 2014.

[64] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. "Shasta: A Low Overhead, Software-only Approach for Supporting Fine-grain Shared Memory". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1996.

[65] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. "Fine-grain Access Control for Distributed Shared Memory". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1994.

[66] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2018.

[67] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. "Distributed shared persistent memory". In: *ACM Symposium on Cloud Computing (SoCC)*. 2017.

[68] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. "StRoM: smart remote memory". In: *European Conference on Computer Systems (EuroSys)*. 2020.

[69] "std::weak_ptr". In: https://en.cppreference.com/w/cpp/memory/weak_ptr (last accessed on 10/15/2020).

[70] "Stingray SmartNIC Adapters and IC". In: https://www.broadcom.com/products/ethernet-connectivity/smartnic (last accessed on 10/15/2020).

[71] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. "DaRPC: Data Center RPC". In: *ACM Symposium on Cloud Computing (SoCC)*. 2014.

[72] Andrew S Tanenbaum and Herbert Bos. *Modern Operating Systems*. 2015.

[73] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. "Borg: The next Generation". In: *European Conference on Computer Systems (EuroSys)*. 2020.

[74] "TLC Trip Record Data". In: https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page (last accessed on 10/15/2020).

[75] Po-An Tsai and Daniel Sanchez. "Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019.

[76] Shin-Yeh Tsai and Yiying Zhang. "LITE Kernel RDMA Support for Datacenter Applications". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2017.

[77] "Understanding the Linux Virtual Memory Manager, Chapter 10 Page Frame Reclamation". In: https://www.kernel.org/doc/gorman/html/understand/understand013.html (last accessed on 10/15/2020).

[78] "Validating References with Lifetimes". In: https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html (last accessed on 10/15/2020).

[79] Chenxi Wang, Ting Cao, John Zigman, Fang Lv, Yunquan Zhang, and Xiaobing Feng. "Efficient Management for Hybrid Memory in Managed Language Runtime". In: *Network and Parallel Computing (NPC)*. Edited by Guang R. Gao, Depei Qian, Xinbo Gao, Barbara Chapman, and Wenguang Chen. 2016.

[80] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. "Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2019.

[81] Paul Wilson. "Operating System Support for Small Objects". In: *International Workshop on Object Orientation in Operating Systems (IWOOOS)*. 1991.

[82] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. "The Case for Compressed Caching in Virtual Memory Systems". In: *USENIX Annual Technical Conference (ATC)*. 1999.