A Tale of Two Paths: Toward a Hybrid Data Plane for Efficient Far-Memory Applications

Lei Chen, *University of Chinese Academy of Sciences;* Shi Liu, *UCLA;* Chenxi Wang, *University of Chinese Academy of Sciences;* Haoran Ma and Yifan Qiao, *UCLA;* Zhe Wang and Chenggang Wu, *University of Chinese Academy of Sciences;* Youyou Lu, *Tsinghua University;* Xiaobing Feng and Huimin Cui, *University of Chinese Academy of Sciences;* Shan Lu, *Microsoft Research;* Harry Xu, *UCLA*

https://www.usenix.org/conference/osdi24/presentation/chen-lei

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

A Tale of Two Paths: Toward a Hybrid Data Plane for Efficient Far-Memory Applications

Lei Chen[†*] Shi Liu[ψ*] Chenxi Wang[†] Haoran Ma[ψ] Yifan Qiao[ψ] Zhe Wang[†]
Chenggang Wu[†] Youyou Lu[‡] Xiaobing Feng[†] Huimin Cui[†] Shan Lu[θ] Harry Xu[ψ]
University of Chinese Academy of Sciences[†] *UCLA*[ψ] *Tsinghua University*[‡] *Microsoft Research*[θ]

Abstract

With rapid advances in network hardware, far memory has gained a great deal of traction due to its ability to break the memory capacity wall. Existing far memory systems fall into one of two data paths: one that uses the kernel's paging system to transparently access far memory at the page granularity, and a second that bypasses the kernel, fetching data at the object granularity. While it is generally believed that object fetching outperforms paging due to its fine-grained access, it requires significantly more compute resources to run object-level LRU and eviction.

We built Atlas, a hybrid data plane enabled by a runtime-kernel co-design that simultaneously enables accesses via these two data paths to provide high efficiency for real-world applications. Atlas uses *always-on* profiling to continuously measure page locality. For workloads already with good locality, paging is used to fetch data, whereas for those without, object fetching is employed. Object fetching moves objects that are accessed close in time to contiguous local space, dynamically improving locality and making the execution increasingly amenable to paging, which is much more resource-efficient. Our evaluation shows that Atlas improves the throughput (*e.g.*, by 1.5× and 3.2×) and reduces the tail latency (*e.g.*, by one and two orders of magnitude) when using remote memory, compared with AIFM and Fastswap, the state-of-the-art techniques respectively in the two categories.

1 Introduction

Today's datacenters commonly suffer from low memory utilization [21]; yet, datacenter applications are increasingly memory-constrained [19, 36, 42, 62] due to their need to hold large datasets in memory for quick data analytics [11, 76] or machine learning [8, 53]. Thanks to the high bandwidth and low latency provided by modern network fabrics such as InfiniBand, far memory techniques [9, 25, 57, 67–69] enable an abstraction of unlimited memory for applications by allowing them to use available memory on remote servers, thereby simultaneously improving application performance and datacenters' overall memory utilization.

Although techniques such as RDMA enable fast network accesses, each remote access is still at least an order of magnitude slower than a local access. As such, it is paramount to optimize the remote access data plane so that applications can benefit from increased memory capacity without suffering a significant performance hit. A major line of work for accessing remote memory is using the kernel's paging system, exemplified by techniques such as InfiniSwap [25], Fastswap [9], Canvas [68] and Hermit [55]. These techniques allow applications to transparently access far memory at the *page* granularity, using the kernel's swap system to swap pages in and out between local and remote memory.

While paging works well for applications that perform bulk data movement and exhibit clear (sequential or strided) access patterns, its coarse granularity incurs substantial *I/O amplification* (*i.e.*, pages loaded only contain a small amount of useful data) for applications that exhibit irregular (or random) access patterns, such as Memcached [5] and graph applications [34]. To reduce I/O amplification, a recent line of work exemplified by AIFM [57] and Kona [13] advocates to access data at a much finer (object) granularity using a user-space runtime system. Swapping objects, rather than pages, can significantly reduce the amount of useless data swapped, leading to higher efficiency. Furthermore, since objects are the data abstraction for developers to write programs, they carry semantics (*i.e.*, user intention) that can be exposed to and used by the runtime to perform additional optimizations, such as data-structure-based prefetching.

Fetching objects at runtime, however, comes at a cost. A drawback that was often overlooked by existing works is that object fetching requires *non-trivial compute resources* to profile object usage, identify patterns, and perform object-level LRU and eviction. For instance, running an object-level LRU algorithm is **one order of magnitude** more expensive than page-based LRU due to a huge number of objects to be processed and the lack of hardware support for tracking object accesses. This overhead is significantly more pronounced in real-world scenarios where CPUs are all busy with executing application threads—given a tight time budget, memory

* Contributed equally.
Corresponding authors: Chenxi Wang and Harry Xu.

management threads cannot scan enough objects to make accurate LRU and eventually have to evict arbitrary objects.

As a result, the right access mechanism is essentially the result of a tradeoff between program locality (*i.e.*, how bad I/O amplification can be) and the amount of compute resources available (*i.e.*, how many cores can be dedicated to object-level memory management tasks). For programs with poor locality, the overhead of object-level memory management can be offset from the large gains of reducing I/O amplification. On the other hand, for programs with good locality and insignificant I/O amplification, the overhead of object fetching stands out, especially in an environment where applications have taken all compute resources (see §3).

There is a recent line of compiler-based techniques (as exemplified by Mira [26]) that profile a program *offline* to understand such a tradeoff, so that compiler can statically choose the mechanism for each data access when compiling the program. However, offline profiling hinges upon program input. For interactive applications such as Memcached, their input data comes from users and keeps changing, rendering a dry-run-based technique ineffective.

Major Insight. The main question we ask in this paper is: can we enable *always-on* profiling for an application to identify its access patterns and dynamically switch between paging and object fetching to adapt to the observed patterns? This approach, if implemented efficiently, has two advantages over the state-of-the-art techniques. First, its continuous profiling identifies patterns *on-the-fly* for different computation stages or parallel threads accessing different data structures, even if the program input keeps changing. As a result, it can quickly change the access path to use a more efficient fetching mechanism. Second, for programs with irregular patterns, object fetching moves objects that are accessed close in time into contiguous memory space, dynamically improving locality as the program executes. This makes it possible for the execution to *be increasingly amenable to paging*, which has higher resource efficiency (see §3).

Although promising, realizing this insight requires overcoming three major challenges, as elaborated below:

The *first challenge* is how to continuously and accurately profile an application with low overhead. Kernel-based page-level profiling, though efficient, does not provide sufficient information with respect to fine-grained data locality. For example, if one single hot object on a page keeps getting accessed but none of other objects do, the kernel-based profiling would identify the page as a hot page although the page clearly possesses poor locality and its accesses should go through object fetching, not paging.

To enable fine-grained profiling, Atlas divides a page into a set of *cards*, each of which is a unit for our locality measurement. We leverage the runtime (and in particular, a *read barrier*) to compute a *card access table (CAT)* (§4.3) for each page, which is a bitmap where each bit corresponds to a card (*i.e.*, consecutive 16 bytes) on the page and a set bit represents that the card has been accessed since the page was allocated or last swapped in. A page with a high *card access rate* (CAR, measured as the percentage of the set bits in its CAT) is deemed to possess good locality and should be accessed with paging, while a page with a low CAR has poor locality and should be accessed with object fetching.

The *second challenge* is how to dynamically switch access mechanisms. Atlas uses a read barrier at each smart pointer dereference. The barrier quickly checks a per-page *path selector flag* (PSF) for the remote page to be accessed. Each PSF is a 1-bit flag, set to either `runtime` or `paging`. `runtime` indicates that the runtime path should be used to fetch individual objects (like AIFM), while `paging` means that the paging path is taken to fetch an entire page. The PSF of a page is updated only when the page is evicted based upon the page's CAR—it is set to `runtime` if the page's CAR is low, indicating the page exhibits poor locality, and `paging` otherwise, indicating good locality.

Although Atlas supports both object fetching and paging at *ingress*, it evicts data only at the page granularity at *egress*, to reduce the high overhead associated with object-level profiling and LRU. While evicting pages may introduce I/O amplification for workloads with poor locality, this impact is insignificant under Atlas, because accesses in these workloads would likely go through the object fetching path, which improves locality by moving objects accessed close in time into contiguous local space. The enhanced locality effectively mitigates the negative impact of page-level eviction.

To reduce fragmentation resulting from dead objects, Atlas runs *concurrent evacuation* tasks that periodically move live objects into contiguous memory space. During each evacuation, Atlas groups recently-accessed objects into contiguous pages to further improve data locality.

The *third challenge* is how to synchronize the two access paths. Since the kernel and the runtime are not coordinated (*e.g.*, the kernel does not inform the runtime of the start or the completion of a page-fault handling), special care must be taken to prevent the two access paths from creating inconsistent data copies. In particular, correctness issues may arise from a set of ingress and egress events (*i.e.*, object-in, page-in, and page-out) that occur simultaneously. Atlas solves the problem with a synchronization protocol (see §4.2), implemented with a combination of runtime and kernel support.

Results. We have evaluated Atlas with a set of eight applications that cover a full range of memory access patterns: sequential, random, and mixed. Our results show that Atlas enables these applications running on remote memory to achieve an overall of 1.5× and 3.2× throughput improvement, compared with AIFM [57] and Fastswap [9], respectively. Atlas reduces the tail latency by one and two orders of magnitude when compared with AIFM and Fastswap. Atlas is available at https://github.com/wangchenxi7/Atlas.

2 Background on Object Fetching

Object fetching is motivated by two observations on the inefficiencies of paging. First, fetching data at the page granularity often leads to I/O amplification [13]. Second, managing data in the kernel space is agnostic to program semantics, resulting in missed optimization opportunities [57, 65, 67]. As such, work has been proposed to manage data with a language runtime at a finer-grained object (or cache-line) granularity [13, 43, 57, 66, 67, 69]. Unlike paging, the runtime can only manage objects in user space, which results in two consequences: (1) the runtime must change the virtual address of an object when moving it and hence must change all its pointers; and (2) the runtime must maintain all metadata itself (*e.g.*, LRU), which used to be maintained by the kernel. Here we focus our discussion on AIFM [57]. AIFM proposes two abstractions for developers to manage remote memory: *remoteable pointer* and *dereference scope*.

Remoteable pointer. AIFM extends the *smart pointer* abstraction of C++ to implement remoteable pointers (`RemPtr`) for remote data management. There are two types of `RemPtr`: 64-bit unique remoteable pointers (similar to `std::unique_ptr`) and 128-bit shared remoteable pointers (similar to `std::shared_ptr`). Developers need to explicitly declare data as remote type and manage them via the `RemPtr`. For example, each unique `RemPtr` has 64 bits—the lower 47 bits are used as the virtual address of the data, and the upper 17 bits are used to record metadata, such as dirty (D), present (P), hot (H), evacuated (E), *etc.* When accessing data via a `RemPtr`, AIFM checks the metadata of the `RemPtr` to detect its status, *e.g.*, checking the P bit to see if the object is in local memory. Next, AIFM masks the `RemPtr` to obtain the actual virtual address.

Dereference scope. Each smart pointer dereference and subsequent raw pointer accesses must be enclosed by a dereference scope, which works as an *evacuation fence* to guarantee correctness. AIFM performs periodical *concurrent object evacuation* that swaps out cold objects to remote memory and compacts local memory to improve data locality. It is challenging to move objects when they are being used by other threads since moving objects requires updating all their pointers. Smart pointers solve this problem because these pointers can be recorded in object headers and updated after moves are conducted. However, an application may read raw pointers from smart pointers and store them in registers or on the stack, which cannot be updated by the runtime.

To guarantee correctness for pointer updating, AIFM requires developers to explicitly declare dereference scopes for each object, which define where raw pointers of the object may exist. Evacuation of the object never happens concurrently with the execution of any of its dereference scopes that started before the evacuation decision. A dereference scope serves as a synchronization mechanism between an event that moves the object and another that uses it.

3 Motivation

We now motivate the necessity of a hybrid data plane. We first demonstrate the diverse memory access patterns of real-world cloud applications and explain the underlying reasons. Next, we compare fetching performance between using a runtime and the kernel's paging system. For the runtime approach, we re-implemented applications with AIFM [57]. For paging, we used Fastswap [9]. Finally, we discuss the opportunities provided by *dynamic* path switching.

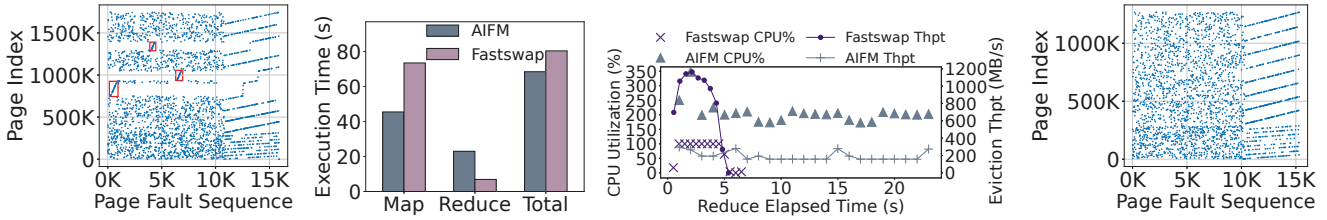
Diverse memory accesses. Real-world applications exhibit complicated memory access patterns, which are a combination of multiple primitive patterns such as sequential, strided, skewed, and random. Access patterns depend on at least two factors: (1) the computation model and (2) the data model. Next we elaborate on these factors:

On one hand, many applications are phase-changing and each phase follows a distinct computation model. On the other hand, the same phase may exhibit varied access patterns when processing different data structures.

An example is data-processing applications [76, 78] that implement MapReduce. We experiment with Metis [44], a MapReduce framework optimized for multicore architectures, with a Page View Count (PVC) program [29, 56] and report its page fault sequence in Figure 1(a). Since PVC is executed with 8 cores, we launch 8 threads for each (Map or Reduce) phase to exploit data parallelism. During the Map phase, each thread loads chunks of input data from the disk and initializes loaded website URLs and users as memory data. Next, PVC shuffles URLs into different buckets of a hash table based on their hash values. The Reduce phase scans each entry to count each URL's users.

The left/right part of Figure 1(a) illustrates the page fault sequence of the Map/Reduce phase. The Map phase (left) inserts URLs into the hash table, and accesses there are mostly random. However, given that the dataset used to run this program is *skewed*, there are several ranges of sequential accesses in the Map phase, as highlighted in the boxes (*i.e.*, certain hash buckets are much larger than others and hence traversing these buckets exhibits sequential patterns). During the Reduce phase (right), each task that aggregates users of URLs scans entries in a bucket sequentially, resulting in a clear sequential access pattern, as shown in the second (right) half of Figure 1(a).

Granularity-performance tradeoff. Object fetching minimizes I/O amplification by fetching fine-grained objects [14, 57]. However, compared to paging, object fetching does not always show clear benefits—for workloads with good locality, data on the same pages are accessed close in time and the kernel can already effectively and accurately prefetch data. When benefits are insignificant, the overhead for object-level memory management stands out. To compare fetching efficiency between the runtime and the kernel, we run the Metis



(a) Page Fault Trace 1 (b) Throughput (c) Memory Eviction (d) Page Fault Trace 2

Figure 1: Statistics of Metis PageViewCount (MPVC): (a) access patterns, (b) performance comparisons between AIFM and Fastswap, (c) comparisons of eviction throughput (dotted lines) and CPU usage (crosses and triangles) between AIFM and Fastswap, and (d) access patterns when input is changed to Wikipedia Italian [6]. For these experiments, 25% of the working set resides in the compute server's local memory. Sequential accesses (due to skewness) in the Map phase are highlighted in red boxes in (a), while in (d) such patterns do not exist.

PVC benchmark on AIFM and Fastswap, respectively. Figure 1(b) reports their performance comparisons.

Since a MapReduce program has clear phases, we broke down the execution time into Map and Reduce. AIFM outperforms Fastswap by 1.6× in the Map phase due to object fetching—most remote accesses in Map are random as words are inserted into different buckets of the hash map. On the contrary, AIFM underperforms Fastswap by 3.3× in the Reduce phase, which exhibits clear sequential patterns.

Object eviction cost. The main reason why object fetching underperforms paging for programs with good locality is the high cost associated with profiling objects and maintaining object-based LRU for eviction. For example, eviction must be done quickly as it blocks further memory allocations [55]. As a result, AIFM constantly maintains dozens of profiling/eviction threads to track the hotness of (billions of) objects and evict cold objects. However, if these threads cannot obtain enough CPU resources from the application, they end up scanning only a small percentage of objects before time runs out and then evict objects with limited hotness information, resulting in data thrashing (*i.e.*, hot objects get swapped out and quickly swapped back in).

Figure 1(c) compares the eviction throughput and CPU utilization for eviction of AIFM and Fastswap during the Reduce phase. AIFM continuously performs object-level hotness tracking and eviction with around 200% (up to 350%) CPU usage in the entire Reduce phase. On the contrary, Fastswap finishes most of the page eviction task within the first five seconds and consumes no more than 100% CPU resources during the eviction. Overall, Fastswap consumes *an order of magnitude* less compute (cycles) than AIFM for eviction over the Reduce phase. Even with significantly fewer CPU resources, Fastswap's eviction throughput is still **∼5**× higher than that of AIFM, due to the low memory management cost associated with paging.

Necessity of online profiling and path switching. Offline profiling techniques [26, 37, 41, 54, 70] were proposed to analyze program semantics and data accesses. However, these techniques are ineffective in identifying the optimal solution for a real-world application for two major reasons.

On the one hand, even if the application's computation phases may be analyzed by an offline profiling technique, its access patterns can change dramatically in response to *inputs*. As Figure 1(d) demonstrates, when fed with a different dataset (which does not exhibit skewness), the program's access patterns change significantly—*e.g.*, due to the lack of skewness, the Map phase no longer exhibits sequential patterns. In fact, for any interactive applications including Memcached [5], DataFrame [46], or streaming data systems [17, 34, 64], their behaviors and access patterns vary significantly with different user requests and workloads.

On the other hand, as discussed earlier, object fetching consumes extensive CPU resources. This may be acceptable when CPU resources are not fully saturated but becomes problematic as soon as all CPU cores are occupied (*e.g.*, another tenant starts using the server). Clearly, offline profiling is not able to predict such environmental changes.

These issues necessitate a dynamic technique that can continuously profile program executions and perform runtime data path switching as new behaviors and/or environmental changes are detected. Our main objective is to use object fetching to minimize I/O amplification and enhance locality, paving the way for subsequent accesses to operate on data with established locality and thus benefit from paging that is considerably more resource efficient.

4 Atlas Design and Implementation

This section presents Atlas's design. Like AIFM, Atlas requires programs to use smart pointers (*i.e.*, to implement barriers) and declare dereference scopes for objects (inspired by C++ weak pointers [4] and Folly RCU guards [1]). Objects are managed by Atlas's hybrid data plane. Atlas can also take the same user-defined programming/offloading hints and object-level prefetching logic as required by AIFM. Atlas uses such hints in the object fetching path.

4.1 Overview

Inspired by the design of the Java heap [51], Atlas divides a page into *cards* to enable fine-grained profiling for accesses. For each page, Atlas builds a card access table (CAT), which is a bitmap where each set bit represents a card that has been accessed since the page was allocated or last swapped in.

CATs for contiguous pages are allocated contiguously in a separate memory space. This design enables not only fine-grained access profiling, but also simple mapping from a virtual address to its CAT entry—this can be done with efficient bit-wise operations on the address. Each card represents 16 consecutive bytes, which provides a fine enough granularity as most objects are at least 16 bytes in our workloads.

Atlas maintains a 1-bit *path selector flag* (PSF) for each page, which works as an indicator of the data path for data access on the page. A `runtime` value indicates that data should be retrieved by the runtime at the object granularity (*i.e.*, runtime path). A `paging` value indicates that data should be paged in by the kernel (*i.e.*, paging path). Atlas updates the PSF of each page to `runtime or paging` at the moment the page is swapped out if its CAR (*i.e.*, the percentage of the set bits among all bits in a CAT) goes below or above a threshold (*i.e.*, 80% used in our evaluation, see §5.4).

Ingress. Atlas uses a *read barrier* that executes at each *smart pointer dereference*. The barrier first checks whether the accessed data is remote. In AIFM, this is done by using a bit in each pointer to encode the location of the referenced object—these pointers are updated once the objects they point to are swapped in or out. Atlas, however, cannot adopt this approach due to the use of the hybrid data plane—when data is *paged* out, Atlas cannot update any pointers. To solve the problem without incurring the cost of checking with the kernel at every read, Atlas leverages *hardware transaction memory* and, in particular, Intel's TSX [31], to run a quick check—Atlas accesses the address in a hardware transaction, which aborts if the address is not on a mapped page.

Upon an abort, the barrier reads the PSF for the page to be accessed and determines which path (runtime *vs.* paging) the access should take. If the runtime path is taken, the object is moved (*i.e.*, address changed) to a local page on the compute server and its pointers are updated; otherwise, the page containing the object is swapped in as a whole and the address of the object remains the same (without requiring pointer updating).

Egress. Given that the majority of the object-fetching overhead comes from the need to find and evict cold objects, Atlas utilizes a single path, *i.e.*, paging, to swap out data. This approach achieves a sweet spot in balancing overhead and benefits—on one hand, it significantly reduces the compute resource usage for object fetching because of the elimination of maintaining an object-level LRU; on the other hand, given that object fetching gradually improves locality (by moving together objects accessed closely in time), the amount of useless data in each swap-out (and thus the I/O amplification) is reduced progressively during execution.

Another reason to not evict objects individually is that it can potentially *hurt locality*—after objects are fetched in, those that were scattered in remote memory but accessed together were moved into contiguous local space; however, these objects may not be evicted at the same time; evicting them individually would make them go to unrelated locations in remote memory, disrupting established locality.

Synchronization. Allowing the two paths to co-exist in harmony requires overcoming the following three synchronization challenges: (1) *ingress synchronization* between object-in and page-in, (2) *egress synchronization* between object-in and page-out, and (3) *move synchronization* between object-in and evacuation. AIFM already solves the third problem with the declaration of dereference scopes, while the other two are unique challenges that we target in Atlas.

4.2 Synchronization of the Two Paths

Atlas builds its object fetching path upon the same two abstractions used by AIFM: the smart pointer (which is an extension of C++ smart pointer) and the dereference scope. This section elaborates on the synchronization mechanism between the object fetching path and the paging path.

```
1  class AtlasUniquePtr<T>{
2    struct AtlasMetadata{
3      unsigned long is_moving : 1;
4      unsigned long access : 1;
5      unsigned long reserve : 2;
6      unsigned long offload : 1;
7      unsigned long size : 12;
8      unsigned long addr : 47;
9    } metadata; // 64 bits
10   AtlasUniquePtr(T* obj);
11   T* get_raw();
12 }
```

Figure 2: Atlas unique pointer metadata.

Pointer Metadata. Before discussing our barrier logic, we first present the format of Atlas pointers, which are built on C++ smart pointers. Atlas uses two types of smart pointers: unique pointers (similar to `std::unique_ptr`) and shared pointers (similar to `std::shared_ptr`). Figure 2 shows the layout of an Atlas unique pointer. These fields are added for the purpose of synchronization and pointer updating.

Each such pointer has 64-bit metadata, in which 47 bits (`addr`) store the object's raw pointer, 12 bits (`size`) record its size, 1 bit (`access`) represents whether the object has been accessed since the last evacuation (which will be used by the evacuator to group recently accessed objects, see §4.3), 1 bit (`offload`) indicates whether a function is being invoked on the object on the remote side, and 1 bit (`is_moving`) indicates whether the object is being moved (*e.g.*, due to evacuation); this bit will be used for synchronization between two threads trying to move the same object. The remaining 2 bits (`reserve`) are reserved for future use. Note that 12 bits can represent a size up to 4KB. Objects larger than that are placed in the huge-object space of the heap for which paging is the only option. `get_raw` retrieves the raw pointer from a smart pointer.

A shared pointer allows aliasing. Atlas treats the first shared pointer of an object as the main pointer. A shared pointer's layout is similar to a unique pointer, except that it has an additional 8 bytes to chain all pointers—when the main pointer is being released, Atlas follows the chain to se-

lect a new main pointer. If an object is referenced by shared pointers, Atlas needs to update all of them (by following the chain).

Developers need to explicitly declare data types with smart pointers. Developers can access data with raw (regular C++) pointers by first retrieving such raw pointers from smart pointers. However, this can only be done within an explicitly declared dereference scope. Figure 3 illustrates an example of retrieving and manipulating data from Atlas smart pointers, confined by a dereference scope. As discussed in §2, dereference scopes synchronize with object migration tasks—once raw pointers are retrieved and actively used, their objects are not allowed to move, and vice versa. Atlas executes a `pre_scope_barrier` and a `post_scope_barrier` at the beginning and the end of the dereference scope, respectively.

```
deref_scope (smart_ptr) {
  pre_scope_barier(smart_ptr); // Algorithm 1
  Data * object = smart_ptr.get_raw();
  /* Operations using the object */
  ...
  post_scope_barrier(smart_ptr); // Algorithm 2
}
```

Figure 3: Dereferencing an Atlas unique pointer in a deref scope.

Synchronization invariants. We present a set of high-level invariants that Atlas maintains to solve the three synchronization problems: (1) preventing an object from being fetched from the two paths simultaneously (object-in *vs.* page-in), (2) preventing pages containing objects that were just runtime-fetched from being immediately swapped out (object-in *vs.* page-out), and (3) preventing an object from being simultaneously runtime-fetched and moved by the evacuator (object-in *vs.* evacuation).

Invariant #1: Object-in vs. page-in. At any moment, all data on the same page must go through the same access path as guided by the page's PSF. In other words, Atlas prohibits scenarios where certain requests are served by paging while others are served by the runtime for the same page. Given that Atlas changes PSF only at page-out (as opposed to setting it while the page is in local memory), such scenarios can never occur and this invariant is guaranteed by design.

Note that there is no issue if two threads fetch the same page from the paging path—the kernel's swap system guarantees only one page can be mapped. Fetching the same object from two threads with the runtime path is not a concern either: it is a solved problem in the literature of moving garbage collectors [43] where pointer updating is used as a synchronization point and only one object is retained.

Invariant #2: Object-in vs. page-out. Since swap-out events can occur at any time with the runtime path uninformed, Atlas enforces that pages containing objects whose dereference scopes are actively executed cannot be swapped out. This is because if such pages are swapped out before their dereference scopes finish, these objects may be fetched back in immediately from the runtime path, requiring pointer

Algorithm 1: Atlas Pre-Scope Barrier (Simplified).

```
   /* derefcnt > 0 precludes the page's swap-out     */
1  atom_inc(find_page_meta(addr).derefcnt)
2  if not tsx_check_local(addr) then    /* Remote object */
3      if take_runtime_path(addr) then  /* Runtime path */
4          new_addr ← find_addr(addr, this.size)
           /* Inc/dec the new/old page's derefcnt     */
5          atom_inc(find_page_meta(new_addr).derefcnt)
6          atom_dec(find_page_meta(addr).derefcnt)
7          alloc_copy_update(addr, new_addr, this.size)
8          this.metadata.addr ← new_addr
9          addr ← new_addr
10     end
11     else                                  /* Paging path */
12         * (char*) addr
13     end
14 end
```

Algorithm 2: Atlas Post-Scope Barrier.

```
1  atom_dec(find_page_meta(this.addr).derefcnt)
```

updating. Pointer updating cannot be done when the raw pointers of these objects are active on the stack. As a result, these pages cannot be swapped out until none of their objects are executing their dereference scopes.

Atlas achieves this by maintaining a per-page *deref count*, which is incremented when any object on the page enters a dereference scope and decremented when the scope finishes. Any page with a non-zero deref count is skipped when the kernel looks for swap-out victims. Note that this does not create much impact on performance because the pages whose objects are actively used are usually hot pages and unlikely to be selected as swap-out victims anyway.

One issue that may arise from this protection is a potential live lock on the object-fetching path: either an ill-defined large dereference scope or many active dereference scopes in a parallel application may potentially lead to too much data getting pinned in local memory, which may result in out-of-memory errors. To tackle this issue, Atlas monitors the pinned data and forces the flipping of their containing pages' PSFs (to use paging) upon memory pressure. Once these pages are swapped out, they will be paged in—this solves the problem as page-in does not need pointer updating.

Invariant #3: Dereference scope vs. evacuation. Evacuation threads may move an object while another thread is executing the object's dereference scope. This must not occur because evacuation requires pointer updating, which cannot be done when a dereference scope is being executed (and raw pointers are used). To this end, Atlas uses the page's deref count to synchronize between evacuation threads and dereference scopes. A non-zero dereference count prevents the page from being evacuated.

Compared to AIFM, Atlas employs a slightly different definition of dereference scope. AIFM chose to decouple dereference scopes from the barrier—it allows one dereference scope to cover multiple smart pointer dereferences, serving

as a coarse-grained fence between application threads and the evacuator. On the contrary, Atlas employs *fine-grained* dereference scopes, each of which is associated with one single smart pointer dereference. This choice was made based on our observation of frequent evacuations; using coarse-grained dereference scopes would require constant synchronizations between application and evacuation threads, leading to performance and latency impact. Fine-grained dereference scopes not only reduce the degree of blocking but also help alleviate potential live locks. Although a finer granularity increases barrier overhead, this overhead is often amortized by a large number of raw pointer accesses and computation within each scope. A detailed overhead analysis can be found in §5.2 and §5.4.

With the invariants discussed above, we proceed to presenting our barrier logic, which is shown in Algorithm 1 and Algorithm 2. As illustrated in Figure 3, Atlas executes Algorithm 1 and Algorithm 2 at the beginning and the end of a dereference scope, respectively.

Pre-scope barrier. Atlas first atomically increments the deref count for the page containing the object (Line 1). This indicates that the page has an object whose dereference scope is being executed, preventing the paging system from swapping out the page (*i.e.*, Invariant #2). This step must be done before the barrier starts to guarantee that (1) if the page is local, it cannot be swapped out from this point on, or (2) if the object is remote, once it is fetched in, its containing page cannot be swapped out.

Atlas uses Intel's TSX [32] to efficiently check if the address `addr` is local. Atlas starts an RTM transaction, which contains nothing but a dereference of the object. If the object's containing page is unmapped, the RTM transaction will abort with a special status captured by Atlas, which verifies the status by checking with the kernel. This hardware-based check is $\sim$14$\times$ faster than a purely software-based approach that relies on a system call that walks the page table and checks whether the page is local based on its PTE. A `true` value (*i.e.*, the object is local) returned by TSX directs the execution to exit the barrier immediately. Otherwise, Atlas checks the PSF corresponding to the address (Line 3) to decide whether this access should take the runtime (Lines 4-9) or the paging path (Line 12).

Using TSX to check object location may introduce false positives—a transaction may abort even if data is local. Since such cases are rare (*e.g.*, less than 1/10000 in our experiments), Atlas takes an optimistic approach to handle them. Upon a TSX abort, Atlas sends an RDMA read to access the remote object and simultaneously issues a page table walk to verify the object's location. If the verification fails (indicating the object is local), the fetched object is discarded. This approach introduces only a negligible overhead (*i.e.*, a small number of unnecessary RDMA reads).

Runtime path. `take_runtime_path` in Algorithm 1 checks the PSF of the page corresponding to `addr` and returns `true` if the PSF is `runtime`, indicating that object fetching should be performed. For ease of presentation, Algorithm 1 is significantly simplified to *not* show details of how to synchronize between threads to guarantee the absence of race condition when multiple threads fetching the same object. Atlas first finds a new address to which the object will be moved (Line 4). Since this address is on a new page, before moving the object, the deref count of the new page must be incremented (Line 5) to ensure that from this point on, the new page cannot be swapped out until the dereference scope finishes (*i.e.*, Invariant #2). The barrier also needs to decrement the deref count of the old page (Line 6) that was incremented earlier in Line 1.

Next, Atlas fetches the object by allocating a new object of the same size (using our log-structured allocator discussed in §4.3), copying the object's data into the new object, and updates its pointers (Line 7). Atlas subsequently changes the `addr` field of the pointer to the new address (Line 8). Pointer updating is done by retrieving the object's pointer from its header and updating their addresses, in a way similar to how it is done in AIFM. If it is a shared pointer, all other pointers will be retrieved from the main one and updated accordingly. The object's `is_moving` field is used to synchronize between pointer updating events performed by multiple threads. The synchronization details are omitted for simplicity. After the object is moved to a local page, future accesses to the object will follow the PSF of the new page.

Paging path. The paging path simply touches the object (Line 12) to ensure that the page fault handling is *completed* after the execution passes this line.

Post-scope barrier. The post-scope barrier has much simpler logic, as shown in Algorithm 2. All it needs to do is to atomically decrement the page's deref count, indicating the finishing of the dereference scope. When its deref count becomes zero, this page is subject to swap-out again (*i.e.*, Invariant #2).

4.3 Memory Management

Atlas's heap is composed of a *normal-object* space, a *huge-object* space, a *metadata* space, and an *offload* space. Atlas manages the normal-object space via a log-structured allocator [57, 58] and maintains a background evacuator to reduce fragmentation by compacting live objects. Atlas does not handle huge objects that cannot fit into a page, placing them into the huge-object space and delegating their management to the kernel directly since they are too large to benefit from object-level management. Metadata such as CATs are accessed by both the runtime and paging system, and hence, it is shared between the user and kernel space. The offload space stores objects whose functions can be offloaded to the remote side. We will discuss it shortly.

Object allocation. The log-structured allocator maintains thread-local allocation buffers (TLAB) to reduce the global lock contention during parallel object allocation. The TLAB

is managed at the granularity of log segment which is aligned with a page to guaranteed that no object can go cross the page boundary. Atlas allocates objects contiguously on the TLAB as prior research [65, 70] shows that objects allocated close in time exhibit similar usage patterns. In doing so, objects with temporal proximity are naturally grouped into the same log segment (page), enhancing locality.

Metadata allocation. Metadata such as dereference counters and card tables is allocated in a dedicated metadata space. Atlas maintains a card table for each page to record the object access information. Each card table is a bitmap where each bit represents a consecutive range of 16 bytes. Our experiments show that the sizes of most objects are larger than 8 bytes, making 16 bytes a natural choice for the card size. Each card table is allocated and initialized during the allocation of a log segment. It is freed along with the log segment. The space needed by the card tables is 1/128 of the total memory. In summary, the space overhead is less than 2%.

Object evacuation. The log-structure allocator [58] supports defragmentation via a copying-based evacuator, a technique widely used in modern garbage collectors [20]. In Atlas, we extend the evacuator to improve the temporal locality of pages by grouping hot objects into contiguous log segments (pages) during the evacuation. The evacuator runs concurrently with the application to reduce fragmentation.

The evacuator periodically scans log segments and evacuates a log segment with a high garbage ratio by copying its live objects to a newly allocated target segment. As a result, the target segment is free of fragmentation, and the source log segment can be freed right away. When moving an object, the evacuator maintains its corresponding card table values, *i.e.*, if the object was recently accessed on the source page, the evacuator marks its card bit on the target page during evacuation. Furthermore, Atlas improves evacuation efficiency by prioritizing log segments in local memory and delaying the processing of remote log segments until they are accessed or the free space runs out [67].

The Atlas runtime tracks whether an object has been accessed since the last evacuation via the `access` bit in the smart pointer (see Figure 2). This bit is set by the read barrier when the object is dereferenced and cleared by the evacuator at the end of each evacuation. The evacuator segregates objects that have been accessed since the last evacuation into a set of contiguously allocated log segments. We found this approach to be particularly effective in improving temporal locality for real-world workloads with *skewness* (*e.g.*, 90% of accesses hit 10% objects). The `access` bit allows Atlas to distinguish hot and cold objects in such workloads, leading to a substantial performance boost. Note that this operation is significantly more efficient than maintaining an object-level LRU for eviction. As opposed to ranking objects based on hotness, Atlas's `access` bit simply serves as an evacuation location indicator. Its functionality is similar to CAT but used differently; CAT is read and cleared by the kernel at page eviction while the `access` bit is read and cleared by the runtime at evacuation.

Computation offloading. As shown in many existing far-memory systems, such as Semeru [66], Mako [43], AIFM [57], and Mira [26], offloading memory-intensive operations to the remote side can effectively reduce the data movement overhead. A unique challenge for Atlas is how to enable offloading when paging is used. Under paging, remote memory is managed as a swap partition of a set of swap slots. These slots are agnostic about the remote server's memory addresses. Pointer addresses contained in a page are with respect to the compute server while the page can reside at a completely different address on the remote server. This address mismatch precludes the correct execution of a function on an object directly on the remote server.

To solve the problem, Atlas uses an approach that is similar to Semeru [66]—we reserve a dedicated offload space in the heap. Developers need to explicitly define remoteable data structures and functions (which are similar to those in AIFM). Objects registered as *remotable* are all allocated into this space. Pages in this space have guaranteed virtual address alignment between the compute and remote servers—we modify the paging system to ensure that a page at a virtual address A on the compute server is guaranteed to be still at address A on the remote server when evicted. Atlas requires users to guarantee a remotable data structure cannot reference a non-remotable object. This property ensures address consistency when a function is called remotely.

The offload space is an *object-in, page-out* space, which allows objects to be fetched only through the runtime. This is due to the need to synchronize between the servers for safe remote execution. When a remote function is being invoked on an object, the `offload` field in its smart pointer is used for synchronization—the runtime can not fetch the object until the remote function is finished (and the `offload` bit is cleared). Remotable objects can only be fetched into the offload space to ensure the above-stated properties.

5 Evaluation

5.1 Setup and Methodology

We wrote 7,675 lines of C/C++ code to implement Atlas's runtime library, and added support in the Linux kernel (version 5.14-rc5) for page management (*e.g.*, path synchronization). We ran experiments with one compute server and one memory server connected by a 200 Gbps Infiniband switch. Each server has 2 Intel Xeon Gold 6342 CPUs (24 physical cores each), 256 GB of memory, and a 100 Gbps Mellanox ConnectX-5 InfiniBand adapter. All evaluated systems ran on Ubuntu 18.04. We configured the servers following common practice for low latency [52], disabling Turbo Boost, CPU frequency scaling, and transparent huge pages.

Baselines. Atlas was implemented based on Fastswap and AIFM. For the paging path, Atlas uses unmodified Fastswap with added tasks of profiling and synchronization. For the runtime path, Atlas uses AIFM's ingress algorithm and paging at egress. For evaluation, we used AIFM [57] and Fastswap [9] as our baselines for object fetching and paging, respectively. For Fastswap, we ran the original applications to avoid unnecessary runtime overhead. For AIFM, we used the performance-tuned versions of applications, where all optimizations were enabled including per-thread access pattern tracking, object hotness tracking, and non-temporal programming hints [57]. We turned off offloading when evaluating throughput and latency, leaving its evaluation to §5.4.

Workloads. As shown in Table 1, we evaluated six real-world applications and two synthetic applications, including Metis [44]—an optimized MapReduce framework for multicore architectures, Aspen [17]—a purely functional tree-based graph processing framework, GraphOne [34]—a data store for real-time analytics on evolving graphs, as well as Memcached [5]—an in-memory key-value store. We ran Memcached with two different workloads: a real-world workload (MCD-CL) that comes from Meta's cache system CacheLib [12] and a synthetic workload (MCD-U) generated by YCSB [15] that follows a uniform distribution. We also employed two synthetic applications developed by AIFM's authors to compare Atlas and AIFM. These applications include one batch application, DataFrame [46], and one latency-critical application, WebService.

Covering a wide spectrum of domains and memory access patterns (*i.e.*, sequential, random, skewed, and mixed patterns), these applications can be divided into four categories:

First, both Memcached workloads exhibit random access patterns, leading to significant I/O amplification under paging. The real-world workload MCD-CL has a high level of skewness with *churn* behaviors. *Churn* refers to the phenomenon that hot data in the working set changes rapidly over time. On the contrary, the synthetic workload MCD-U demonstrates completely random behaviors, with no skewness and hot data. As a result, MCD-CL is more amenable to Atlas's dynamic locality improvement than MCD-U.

Second, GraphOne and and Aspen are evolving graph systems, which are representatives of applications that perform analytics over frequently updated datasets. GraphOne uses adjacency lists and edge lists to store an input graph while Aspen utilizes compressed purely-functional trees to store a graph, which supports a higher update rate. The working sets of these applications change continuously. Their accesses are very complex: the first stage builds the graph in memory, exhibiting a random pattern. The second stage runs iterative algorithms where the first iteration does not have locality and thus performs random accesses; the subsequent iterations would enjoy better locality if it runs on Atlas, which dynamically improves the locality during the first iteration. However, updates to the input graph disrupt the locality and hence there can also be many random accesses in the middle of the iterations. We used these two graph frameworks to evaluate how well Atlas can dynamically adjust the data layout and improve locality.

Third, Metis (MapReduce) and DataFrame represent bulk data processing systems with clear phase-changing behaviors (discussed in §3). These workloads are used to evaluate whether Atlas can accurately recognize access patterns and switch to the proper data path. DataFrame is additionally used to evaluate compute offloading due to its memory-intensive operations (§5.4).

Finally, WebService is an interactive web application exhibiting mixed access patterns, from random, pointer-chasing, to sequential accesses.

For Atlas to run these applications, we modified 263 lines of code for Metis, 278 lines for Aspen, 219 lines for GraphOne, and 391 lines for Memcached; the additional code was used to declare smart pointers and dereference scopes. It took one developer a few hours to port each program.

Memory setup. Each application was run with five local memory configurations: 13%, 25%, 50%, 75% and 100%, each representing a specific percentage of an application's working set that can fit into local memory. These configurations were enforced using `cgroup`. The first four configurations were employed to evaluate the performance of the three systems when using different amounts of remote memory, while the 100% (all local memory) configuration was used to assess the runtime overhead of Atlas and AIFM, introduced by the barriers (for smart pointer dereferencing), dereference trace recording (for object-level prefetching), and evacuation (for defragmentation), as well as other bookkeeping overheads; see Table 2 for more details.

5.2 Throughput

We first measured the throughput of the applications with varying local memory ratios. Overall, Atlas outperforms Fastswap and AIFM, respectively, by 3.2× and 1.5×, over the eight real-world applications using remote memory (from 13% to 75% local memory). When running locally (100% local memory), Atlas and AIFM incur an overall overhead of 19.1% and 14.0%, respectively, of which 10.2% and 2.3% are from the barriers. This section reports the overall performance and runtime overhead. We show a detailed overhead breakdown in §5.4.

MCD-CL and MCD-U. Both workloads were configured with the same operation ratios, *i.e.*, 87.4% get and 12.6% set. As shown in Figure 4(a), for a highly-skewed workload like MCD-CL, both Atlas and AIFM outperform Fastswap (by 6.4× and 3.2×, respectively). The performance difference comes primarily from the reduced I/O amplification—Fastswap fetches 26× and 30× more data than Atlas and AIFM, respectively, resulting in wasted memory (for storing unused data) and significantly more swaps. Under 100% local memory, Atlas and AIFM introduce an over-

Application	Dataset	Size	Characteristics
Memcached CacheLib [5] (**MCD-CL**)	Meta CacheLib [12]	50M records	Skewness with churn
Memcached Uniform (**MCD-U**)	Synthetic, uniform distribution [15]	50M records	Random access
GraphOne PageRank [34] (**GPR**)	Twitter 2010 [35]	1.5B Edges, 41.7M Vertices	Evolving graph
Aspen TriangleCount [17] (**ATC**)	Friendster [73]	1.8B Edges, 65.6M Vertices	Evolving graph
Metis Word Count [44] (**MWC**)	The News Crawl Corpus [72]	5.1GB	Phase-changing
Metis PageViewCount (**MPVC**)	Wikipedia English [6]	15GB	Phase-changing with mixed patterns
DataFrame [46] (**DF**)	NYC Taxi [3]	16 GB	Phase-changing with offloading
Web Service [57] (**WS**)	Synthetic [57]	10GB hashmap, 16GB array	Mixed patterns with offloading

Table 1: Applications used for our evaluation.

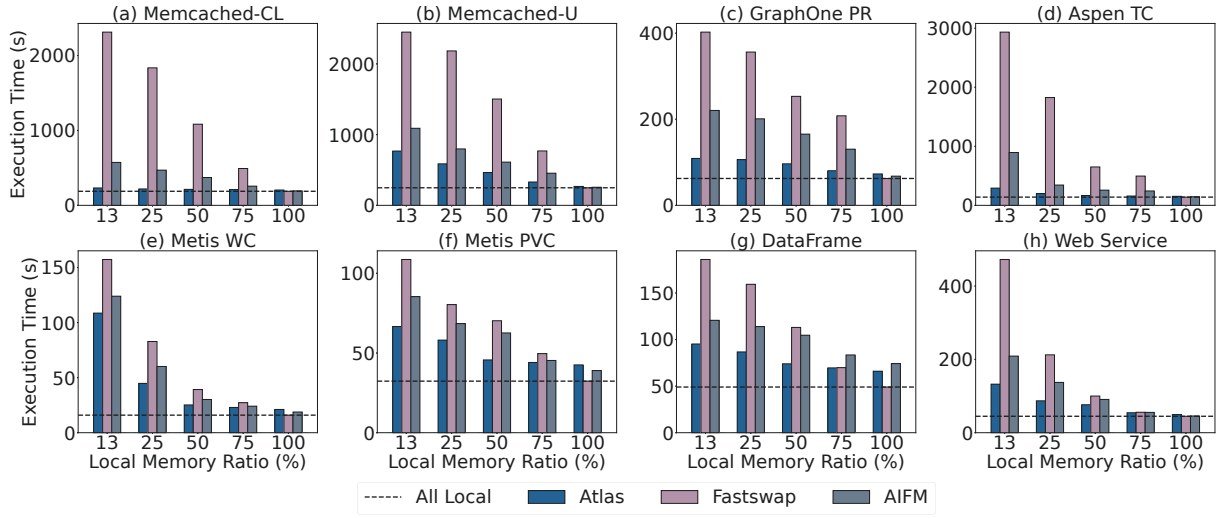


Figure 4: Throughput comparison between Atlas, Fastswap and AIFM with varying local memory ratios. "All Local" lines represent the performance of unmodified applications under 100% local memory.

all overhead of 9.0% and 3.2%, respectively, compared to Fastswap. The primary source of the overhead is the barriers, taking 6.2% and 1.5% of the execution time, respectively. Given that Memcached spends a substantial portion of its execution on communication, the barrier overhead, which is associated with the in-memory processing, is insignificant.

Compared to AIFM, Atlas further improves the performance by 1.2×, 1.8×, 2.2×, 2.5×, under the four different memory configurations (75%, 50%, 25%, and 13%). This improvement stems from a much higher eviction throughput (on average 4.6× higher) in Atlas due to the elimination of object eviction. In addition, Atlas's concurrent evacuator (§4.3) improves the temporal locality by segregating hot objects into contiguous pages, leading to an overall of 18% more accesses that go through the paging path (§5.4). This result was achieved when AIFM used 20 eviction threads while Atlas only used one single swap-out thread in the paging path. MCD-U performs random accesses with no hot data, hindering opportunities for Atlas to improve locality. Hence, the usefulness of the hybrid data plane is limited. However, Atlas still outperforms AIFM by up to 1.4× due to more efficient eviction, as shown in Figure 4(b).

GPR and ATC. To execute an evolving graph engine, we divided the input datasets [35] into three batches, which are incrementally fed to the graph engine. For each batch, the graph engine conducts the following three steps: load the updates, update the graph, and execute the analytics.

As Figure 4(c) shows, in the presence of remote memory, Atlas outperforms AIFM and Fastswap by an average of 1.8× and 3.1×, respectively, on GPR. As stated earlier, graph updating and the first iteration of analytics exhibit random access patterns. As such, GPR's throughput under AIFM is 1.7× higher than under Fastswap. For Atlas, when the analytics starts, objects are accessed and reordered by the object fetching in the first few iterations; in the subsequent iterations, pages storing edge objects are switched to using the paging path due to the gradually established locality. As a result, up to 82% of pages have their PSFs changed during the execution (from object fetching to paging), as demonstrated in Figure 7(b). This improves the analytics throughput.

ATC's computation stages and access patterns are both similar to those of GPR. For ATC, the trees storing the graph data are dynamically reorganized by Atlas's runtime path, leading to ∼38% of pages changing their PSFs (from object fetching to paging). In addition, evacuation improves locality by segregating hot objects from these trees into a few pages, reducing remote memory accesses by 24%. As demonstrated in Figure 4(d), ATC's overall throughput is 2.0× higher under Atlas than under AIFM.

When running on 100% local memory, Atlas's barrier overheads for both GPR and ATC are modest, 8.2% and 4.3%, due to the high ratio between raw pointer accesses and smart pointer dereferences. Oftentimes, one object dereference (*e.g.*, obtaining a vertex that contains a series of edges) is followed by dozens of raw pointer accesses (*e.g.*, to individual edges). Each dereference scope contains an average of *21* raw pointer accesses. In addition, for ATC, the barrier overhead is further diluted due to its higher computation and memory access costs (from poor spatial locality).

MWC and MPVC. Figure 4(e) and (f) respectively show the performance of MWC and MPVC. As discussed in §3, MPVC exhibits a two-phase behavior that can benefit from adaptive path switching, leading to a 1.2× and 1.4× improvement, compared with AIFM and Fastswap, respectively. MWC has a similar two-phase behavior with MPVC but exhibits more random accesses in its map phase, resulting in almost no page that can be flipped to `paging`. Compared to AIFM and Fastswap, MWC has 1.2× and 1.5× performance improvement, respectively.

For these two applications, the runtime overhead is relatively high—32.0% (Atlas) and 19.2% (AIFM), under 100% local memory. These two Metis workloads are both memory-intensive—they keep scanning data with high parallelism, leading to both high barrier overhead and profiling overhead (*e.g.*, for card profiling and access trace recording, see §5.4). Atlas's barrier overhead reaches up to 16.1% and 17.4% for MPVC and MWC, respectively, which are about 4× higher than that of AIFM.

DF. DF is a table-structured in-memory data structure with hundreds of columns and millions of rows, popularized in Pandas [48]. Users can slice data in different ways and run various statistics. As Figure 4(g) shows, Atlas outperforms AIFM by 1.2∼1.4× in the four remote-memory settings. We ran a client, developed by the AIFM authors, to conduct a series of *Copy* and *Shuffle* operations on *DF*. Similarly to Metis, *DF* demonstrates clear phase-changing behaviors when processing different operations—a *Copy* operation copies data from a column, exhibiting excellent spatial locality and a clear sequential pattern, while a *Shuffle* operation reorders rows for each column, exhibiting random patterns. Atlas achieves superior performance to AIFM and Fastswap, due to its adaptive access path selection.

AIFM suffers a higher runtime overhead (51.4%) compared to Atlas (34.7%) despite having a lighter barrier. The reason is that AIFM maintains a remote vector on the memory server for every DataFrame vector to support the eviction of individual objects with varied sizes. During the execution, DataFrame vectors keep getting allocated and resized. As a result, the remote data structure also needs to be frequently resized to maintain a valid mapping from local objects to their remote memory locations. Resizing is a heavy operation as it requires allocating memory and moving all existing objects. Therefore, it becomes a major source of overhead, which can take two-thirds of the runtime overhead under 100% local memory. On the other hand, under Atlas, eviction is handled by the Linux kernel at a fixed page size and there is no need to maintain any remote data structures. Note that frequent resizing of data structures was not observed in other applications. For example, for WS, the hash table array is allocated at the start of the application and its size remains fixed throughout the execution.

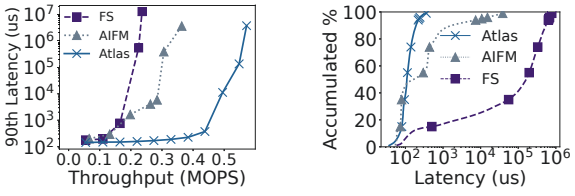
WS. WS is implemented by AIFM's authors to simulate a distributed workload. Each client (thread) sends 32 requests to look up keys in an in-memory hash table and fetches a single 8KB element from an array. This element is then encrypted with Crypto++ [7] and compressed using Snappy [23] before being sent back to the client. We use a 26GB dataset for the evaluation, which is consistent with the dataset used in AIFM [57]. Client requests are generated by following a Zipfian distribution.

As Figure 4(h) shows, compared to AIFM, Atlas improves WS' performance by an average of 1.3× with remote memory. This is due to an extremely large number of objects on the LRU list that must be analyzed by AIFM. AIFM's performance degradation is primarily due to the compute resource contention between application and evacuation threads (discussed in §3), making it hard for evacuation threads to quickly identify and evict cold objects. Consequently, AIFM ends up evicting arbitrary objects to reclaim memory, resulting in data thrashing. By using paging for eviction, Atlas improves the eviction throughput by 5.8×, lifting data eviction efficiency to 5.9 cycles/byte, which is 7.4× higher than that of AIFM (43.7 cycles/byte).

Atlas and AIFM have relatively low overhead for WS due to the coarse-grained data fetching (8KB element) and the subsequent compute-intensive encryption. As a result, Atlas and AIFM introduce a 10.1% and 1.9% runtime overhead under 100% local memory, respectively.

5.3 Latency

This section evaluates the latency distribution using the two latency-critical applications: WS and MCD-CL. The 25% local memory ratio was used in these experiments.



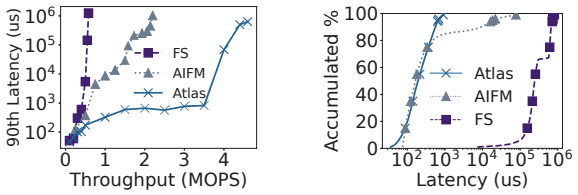
(a) 90th latency-throughput curve. (b) Latency CDF.

Figure 5: (a) 90th latency as a function of throughput; (b) Latency CDF under 0.23 MOPS offered throughput. FS stands for Fastswap.

Web Service (WS). Figure 5(a) compares the tail latency among the three systems. Fastswap's tail latency rapidly grows due to page thrashing caused by severe access amplification. AIFM reduces amplification so that requests are

less blocked by eviction. Despite the reduced amplification, AIFM still has to rank and evict individual key-value pairs, and hence the system saturates at 0.36 MOPS.

Atlas fetches individual key-value pairs initially via the runtime path and places those pairs which belong to the same request together on the same page (because these KV pairs are accessed close in time). As the execution progresses, Atlas switches to paging that can load multiple key-values pairs at the same time. Meanwhile, page-level eviction continuously offers a much higher eviction throughput so that it never blocks swap-ins. As a result, Atlas's tail latency stays low until 0.45 MOPS and can finally reach a peak throughput of 0.57 MOPS. As shown in Figure 5(b), the latencies of AIFM and Atlas are comparable until the 50th percentile, where the application starts accessing many remote objects leading to increased object management overhead. On the contrary, due to the optimized data layout which enables the efficient use of paging, Atlas experiences fewer remote accesses.

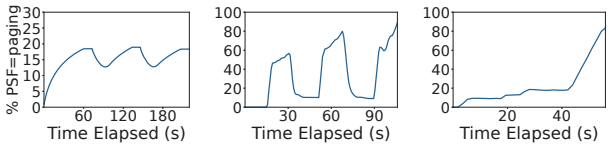


(a) 90th latency-throughput curve. (b) Latency CDF.

Figure 6: (a) 90th latency as a function of throughput; (b) Latency CDF under 1 MOPS offered throughput. FS stands for Fastswap.

MCD-CL. Memcached CacheLib is similar to Web Service as they both access key-value pairs from a hash table. The difference is that every request key in MDC-CL follows a Zipfian distribution, as opposed to accessing key-value pairs always in groups of 32. Figure 6 compares the tail latency among the three systems. It is clear that Atlas outperforms the other two systems. In addition to the same reasons explained above, MCD-CL is a skewed workload and hence a substantial portion (40%) of the improvement comes from the evacuation that groups hot objects in contiguous pages, making these pages amenable to paging.

5.4 Performance Drill Down



(a) MCD-CL (b) GraphOne PR (c) MPVC

Figure 7: The percentage of pages with PSF=`paging` in the memory footprint changes with the elapsed execution time.

Adaptive path switching. To understand the effectiveness of Atlas's adaptive path switching, we measured the percentage of the pages whose PSF is `paging` during the execution. Figure 7 demonstrates how this percentage changes during the execution for three applications: Memcached CacheLib (MCD-CL), GraphOne Pagerank (GPR) and Metis PageViewCount (MPVC). As Figure 7(a) shows, the number of pages that go through the paging path rises and falls over the time due to the *churn* behavior in MCD-CL discussed in §5.1. Since the workload is highly skewed, most accesses fall on a small number of hot objects, which stay in local memory and are moved into contiguous pages (with a high CAR) until the hot spot shifts.

As discussed in §5.1, the execution of GPR has experienced three batches of updates to the input graph, each of which contains two steps: graph building and analytics. During graph building, applying edge-level updates exhibits random access patterns, which can disrupt locality and leave many pages with a low CAR; these pages would have to go through the object fetching path. However, the subsequent analytics (like PageRank) runs multiple iterations; Atlas can quickly improve locality in the first few iterations, making pages turn their PSF to `paging` in subsequent iterations. This pattern can be clearly seen in Figure 7(b).

MPVC has a clear two-phase behavior (see Figure 1(a)) which can be accurately recognized by Atlas —the number of pages that go through the paging path increases dramatically as the phase change is detected by Atlas (shown in Figure 7(c)). To understand the individual contributions of object fetching and evacuation to the locality, we disabled the `access` bit tracking and let the evacuator move live objects without guidance. This reduces the overall percentage of pages that go through paging by 4% on average.

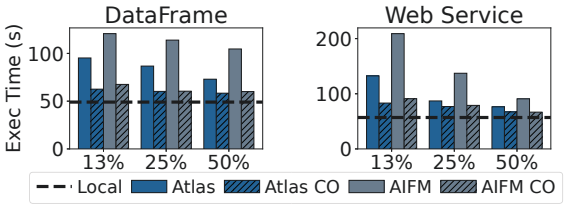


Figure 8: Throughput comparisons of DataFrame (DF) and Web Service (WS) when Atlas and AIFM enable compute offloading. CO stands for variants with compute offloading.

Computation offloading. We compared the offloading performance between Atlas and AIFM using DF and WS. Figure 8 shows the results of Atlas and AIFM with and without offloading. 18 cores were reserved on the remote side for both Atlas and AIFM, which is consistent with the offloading settings used by AIFM [57]. For DF, we offloaded the memory-intensive operations, *i.e.*, `Copy` and `Shuffle`, to the remote side. For WS, we offloaded the heavyweight array processing (on the 16GB data array). Compared to the setting where offloading is disabled (Figure 4 (g) and (h)), the throughputs of Atlas and AIFM are both dramatically improved (by up to 1.5× and 1.9× for DF, and 1.6× and 2.3× for WS, respectively), due to reduced remote accesses and data movement. On the other hand, Atlas and AIFM achieve comparable performance. This is because Atlas focuses on

fetching efficiency; offloading reduces the need for fetching, making Atlas's benefit less significant.

Runtime overhead analysis. To understand the performance penalty introduced by the runtime of Atlas and AIFM, we break down and compare the runtime overhead by sources. When running with all local memory, the runtime overhead of Atlas and AIFM can be divided into five major components, listed in Table 2. Note that the overhead reported here represents the **worst-case scenario** for Atlas when compared against AIFM. When there is remote memory, part of Atlas's runtime overhead can be eliminated by switching to the paging path—dereference trace profiling is not used for paging as its goal is to analyze dereference traces for prefetching objects. Meanwhile, AIFM incurs more profiling overheads that do not exist under the all local memory setting, such as maintaining the object-level LRU for eviction.

Sources of overhead	Functionality	Affected systems
Barrier (Dereferencing)	Correctness guarantee, such as location check & synchronization	Atlas and AIFM
Card Profiling	Offering data path switching hints.	Atlas
Dereference Trace Profiling	Offering object-level prefetching hints	Atlas and AIFM
Evacuation	Defragmentation	Atlas and AIFM
Remote Data Structure Management	Managing object-level eviction	AIFM

Table 2: Major types of runtime overheads, operations involved in each type, and their affected systems.

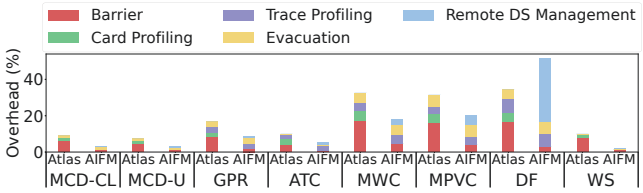


Figure 9: Runtime overhead breakdown: overhead is calculated as the ratio between the extra execution time introduced and the execution time under 100% local memory.

As shown in Figure 9, compared to Fastswap, the extra tasks in Atlas incur a runtime overhead of 7.7-34.7%, while AIFM's overhead is 1.9-51.4%. The overall overheads of the two systems are 19.1% and 14%, respectively. The primary source of overhead for both systems is the barrier (except for DF with AIFM, for which the reasons are explained in §5.2). Specifically, the Atlas barrier accounts for half of the total overhead (∼10%), and its cost is 4.4× of that of AIFM. Note that this overhead correlates with an application's memory access behavior: the most memory-intensive applications suffer the heaviest barrier overhead (MWC, MPVC, DF).

Although Atlas uses a heavier barrier, it underperforms AIFM by *only 4% under 100% local memory*. The reason is three-fold: (1) the barrier overhead is effectively amortized across the computation and raw pointer accesses (§5.2); (2) AIFM's use of coarse-grained dereference scopes leads to higher synchronization costs than Atlas; and (3) there are other operations that also contribute to the runtime overhead. Since the first item has been discussed earlier in this section, here we elaborate on the second and third items.

The barrier conducts two basic tasks, object location checking and synchronization. For location checking, Atlas has a much higher overhead than AIFM due to the use of TSX to detect an object's location whereas AIFM checks a bit on each reference. However, for synchronization, AIFM's coarse-grained dereference scopes incur a higher cost, which effectively reduces the performance gap between the barriers of the two systems. After selecting the victim segments, AIFM's evacuator must wait until all application threads exit their dereference scopes to avoid compacting objects being accessed through raw pointers. This design does not work well for big data applications with high object allocation rates, such as MWC, MPVC and Memcached. On the contrary, Atlas's fine-grained dereference scope design enables evacuation threads to skip the segments (each aligned to a page in Atlas) whose *deref count* is non-zero (indicating they are being used in active dereference scopes) instead of blocking the whole evacuation, leading to significantly reduced synchronization efforts. In fact, Atlas's CPU yield rate caused by synchronization is *an order of magnitude lower* than that of AIFM due to our non-blocking design.

Another major source of overhead is the dereference tracing (to provide prefetching hints), accounting for 14% and 19% of the total overhead for Atlas and AIFM, respectively. Among our applications, DF, MWC, MPVC and GPR use array data structures which are amenable to prefetching. As a result, there is a relatively high tracking overhead (accounting for 34% overhead on average) for both Atlas and AIFM. Other applications such as WS and Memcached use hash maps and small objects as their data structures, which are not as amenable to prefetching as arrays. Hence, for most of their memory accesses, the locations are not tracked and their tracing overhead is much lower. Note that with remote memory, the dereference tracing overhead is significantly lower under Atlas than under AIFM because a large amount of data (*e.g.*, up to 82% for GPR) goes through the paging path, which utilizes the lightweight page-level prefetcher.

CAR threshold. Figure 10 shows the influence of CAR threshold on the throughput of three applications. Picking the right CAR threshold is a tradeoff between fetching efficiency and resource waste. We used 80% as the CAR threshold for flipping PSF in our evaluation. A higher CAR is often too conservative. For example, in the case of MCD-CL, when the threshold is set to 100%, we observed that few pages can be flipped to `paging`. Therefore, most remote objects still have to be fetched individually instead of fetched in batches with page faults, leading to a 25% decrease in throughput. On the contrary, a lower CAR may result in

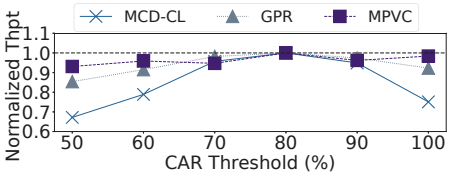


Figure 10: Sensitivity of the CAR threshold.

premature use of paging, leading to I/O amplification. As shown, the best performance is achieved when the threshold is between 80% and 90%. As such, we used the lower bound 80% based on the observation that the bandwidth of a modern network such as InfiniBand [49] is already high and will only become higher in the future, making it possible to transfer (slightly) more data with little overhead.

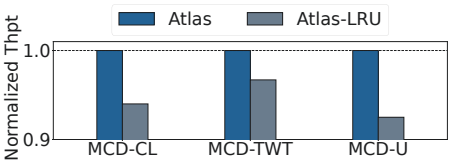


Figure 11: Normalized throughput of Memcached workloads running on Atlas and Atlas-LRU under 25% local memory.

Hotness tracking. Atlas uses an `access` bit on each smart pointer to segregate hot and cold objects during evacuation, offering benefits to workloads that exhibit skewness. We evaluated the effectiveness of Atlas's `access` bit with three skewed workloads, *i.e.*, highly-skewed (Meta, MCD-CL) [2], moderately-skewed (Twitter, MCD-TWT) [74] and uniformed without skewnewss (MCD-U) [15]. We compared Atlas with a baseline (Atlas-LRU) equipped with an LRU-like policy from CacheLib [12], which represents a more accurate approach to identifying hotness.

As shown in Figure 11, Atlas's single-bit design outperforms the LRU-like design by 7.5%, 3.3% and 6.0%, respectively. The LRU-like policy trades compute resources for accuracy by maintaining the logical ordering of objects via a linked list. Each dereference triggers a promotion that moves the object to the head of the LRU list. In order to reduce the overhead, we adopted *flat combining* [30] (to reduce thread lock contention) and ignored the dereferences of an object within 10s (to reduce promotion frequency for extremely hot objects) [12]. However, although an LRU-like policy can reduce the frequency of remote access, it incurs a maintenance overhead of up to 9% due to a huge number of objects.

Of course, the more bits used, the higher accuracy they bring. Atlas allows developers to customize the hotness tracking policy with the two reserved bits in each smart pointer (Figure 2). For our applications, we did not observe significant performance variations between using one and two access bits—likely the ability of distinguishing hot and cold objects is not increased much with two access bits.

6 Related Work

Disaggregation. Resource disaggregation has become a trending architecture for datacenters to improve resource utilization. Its key idea is to break the server hardware boundary and unstrand idle resources of remote servers by leveraging advanced network hardware [22, 28]. Existing systems have demonstrated the viability of disaggregated storage [33, 38], accelerators [50, 63, 75], network [60], and memory [25, 59]. For a memory-disaggregated system, memory spans across multiple servers. The efficient data path of Atlas can speed up the data transfer between servers.

Paging-based far memory. A practical way to deliver far memory is to leverage the paging system to access far memory. Google and Meta have reported their successful deployment of such systems in their datacenters [36, 71]. Many optimizations to the kernel data path have been proposed for improved efficiency, including but not limited to bypassing the block layer [9, 55], prefetching more accurately [45], and reducing interference [68]. The design of Atlas is orthogonal to the underlying paging systems and can directly benefit from optimizations within these systems.

Object-based far memory. Many runtime libraries offer new primitives for object-granularity far memory management, making them a more efficient alternative for scattered data on far memory. For example, AIFM [57] proposed remoteable data structures, FaRM [18] offered key-value interfaces, and Grappa [47] builds a software distributed memory. Atlas focuses on the cooperative use of its two data paths and benefits directly from existing optimizations.

Emerging hardware. Emerging hardware technologies unlock new opportunities for efficient far memory. Clio [27], StRoM [61], and RMC [10] offload functionalities to their customized hardware to reduce network traffic. Finally, CXL [16, 24, 39, 40, 77] and Project PBerry [13, 14] enable far memory access at the cache-line granularity. Atlas directly benefits from the throughput and latency advancements of new hardware technologies. Besides, for hardware solutions with a fixed access granularity, Atlas can improve data locality to improve data transfer efficiency.

7 Conclusion

We present Atlas, a hybrid dataplane that enables efficient far memory for bulk data and scattered objects simultaneously. Atlas outperforms both the state-of-the-art object-based and paging-based far memory systems.

Acknowledgement

We thank the reviewers for their comments and are particularly grateful to our shepherd Malte Schwarzkopf for his feedback. This work is supported by National Key Research and Development Plan of China under grant 2022YFB4500400, National Natural Science Foundation of China under grant 62090024, US National Science Foundation under grants CNS-1763172, CNS-2007737, CNS-2006437, CNS-2106838, CNS-2147909, CNS-2128653, CNS-2301343, CNS-2330831, CNS-2403254, as well as supports from Cisco and Tencent Big Data.

A Artifact Appendix

A.1 Overview

Atlas is a kernel-runtime co-designed system to enable a hybrid remote memory data plane. The artifact includes the custom Linux kernel and the runtime library to enable Atlas-managed applications. To run the artifact, two servers with Intel CPUs connected by InfiniBand are required. The server running the application is the CPU server, while the other server providing remote memory is the memory server. Detailed instructions can be found in Atlas code repository.

A.2 Checklist

- **Hardware:** Two servers with Intel CPUs with TSX, connected by InfiniBand
- **Software Environment:** Ubuntu 18.04, 20.04 or 22.04, with the specified version of MLNX_OFED driver and provided Linux kernel described below
- **Public Link to Repository:** https://github.com/wangchenxi7/Atlas
- **Code License:** MIT License

A.3 Building the Linux Kernel

```
## all operations are performed on both
servers unless specified
cd linux-5.14-rc5
cp config .config
sudo apt install -y build-essential
bc python2 bison flex libelf-dev
libssl-dev libncurses-dev libncurses5-dev
libncursesw5-dev
./build_kernel.sh build
./build_kernel.sh install
./build_kernel.sh headers-install
## edit GRUB_DEFAULT="Advanced
options for Ubuntu>Ubuntu, with Linux
5.14.0-rc5+", or whatever the new kernel
version code is
## edit GRUB_CMDLINE_LINUX="nokaslr
transparent_hugepage=never
processor.max_cstate=0
intel_idle.max_cstate=0 tsx=on
tsx_async_abort=off mitigations=off"
sudo vim /etc/default/grub
sudo update-grub
sudo reboot
```

A.4 Setting up InfiniBand Connection

```
## use Ubuntu 18.04 as an example below
wget https://content.mellanox.com/ofed/
MLNX_OFED-5.5-1.0.3.2/MLNX_OFED_LINUX-5.5-
1.0.3.2-ubuntu18.04-x86_64.tgz
tar xzf MLNX_OFED_LINUX-5.5-1.0.3.2-
ubuntu18.04-x86_64.tgz
cd MLNX_OFED_LINUX-5.5-1.0.3.2-
ubuntu18.04-x86_64
sudo apt install -y bzip2
sudo ./mlnxofedinstall
-add-kernel-support
sudo /etc/init.d/openibd restart
sudo update-rc.d opensmd remove -f
sudo sed "s/# Default-Start:
null/# Default-Start:  2 3 4 5/g"
/etc/init.d/opensmd -i
sudo systemctl enable opensmd
sudo service opensmd start
## assign IPs to InfiniBand interfaces on
both servers
sudo nmtui
```

A.5 Building Atlas Runtime

```
## use gcc-9
cd atlas-runtime/third_party
git clone -depth 1 -b
54eaed1d8b56b1aa528be3bdd1877e59c56fa90c
https://github.com/jemalloc/jemalloc.git
cd ../bks_module/remoteswap
## on memory server
cd server && make
## on CPU server
cd client && make
cd ../../bks_drv && make
cd ../..  && mkdir build && cd build
cmake ..  && make -j
```

A.6 Running Atlas Applications

```
cd atlas-runtime/bks_module/remoteswap
## on memory server
cd server
##./rswap-server <memory server IB ip>
<memory server IB port> <memory pool size
in GBs> <CPU server core count> e.g.,
./rswap-server 172.16.16.1 9999 48 96
## on CPU server
cd client
## edit `mem_server_ip`,
`mem_server_port` and
`SWAP_PARTITION_SIZE_GB` to be consistent
with memory server parameters
vim manage_rswap_client.sh
bash manage_rswap_client.sh install
## run a test
cd atlas-runtime/build/tests/
runtime/unique_ptr
bash test.sh ./unique_ptr_test
```

References

[1] Facebook Folly RCU Library. https://github.com/facebook/folly/blob/main/folly/synchronization/Rcu.h.

[2] Meta CloudLib. https://cachelib.org.

[3] Nyc taxi trips - exploratory data analysis. https://www.kaggle.com/code/kartikkannapur/nyc-taxi-trips-exploratory-data-analysis/notebook.

[4] std::weak_ptr. https://en.cppreference.com/w/cpp/memory/weak_ptr.

[5] Memcached - a distributed memory object caching system. http://memcached.org, 2020.

[6] Konect networks data. http://konect.cc/networks/, 2021.

[7] free c++ class library of cryptographic schemes. https://www.cryptopp.com, 2022.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[9] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[10] E. Amaro, Z. Luo, A. Ousterhout, A. Krishnamurthy, A. Panda, S. Ratnasamy, and S. Shenker. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 3844, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Apache. Apache cassandra. https://cassandra.apache.org, 2021.

[12] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, Nov. 2020.

[13] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, pages 79–92, 2021.

[14] I. Calciu, I. Puddu, A. Kolli, A. Nowatzyk, J. Gandhi, O. Mutlu, and P. Subrahmanyam. Project pberry: Fpga acceleration for remote memory. HotOS '19, pages 127–135, New York, NY, USA, 2019. Association for Computing Machinery.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143154, New York, NY, USA, 2010. Association for Computing Machinery.

[16] Compute express link 3.0. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf, 2022.

[17] L. Dhulipala, G. E. Blelloch, and J. Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019. Association for Computing Machinery, 2019.

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

[19] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *SOSP*, pages 394–409, 2015.

[20] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *PPPJ*, pages 13:1–13:9, 2016.

[21] A. Fuerst, S. Novaković, I. n. Goiri, G. I. Chaudhry, P. Sharma, K. Arya, K. Broas, E. Bak, M. Iyigun, and R. Bianchini. Memory-harvesting VMs in cloud platforms. In *ASPLOS*, pages 583–594, 2022.

[22] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, pages 249–264, 2016.

[23] Google. Google's fast compressor/decompressor. https://github.com/google/snappy, 2020.

[24] D. Gouk, S. Lee, M. Kwon, and M. Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

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

[26] Z. Guo, Z. He, and Y. Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692708, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 417433, New York, NY, USA, 2022. Association for Computing Machinery.

[28] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, pages 10:1–10:7, 2013.

[29] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 260269, New York, NY, USA, 2008. Association for Computing Machinery.

[30] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355364, New York, NY, USA, 2010. Association for Computing Machinery.

[31] Intel Corporation. Transactional Synchronization Extensions. In *Intel® 64 and IA-32 Architectures Software Developers Manual Volume 1: Basic Architecture*, pages 16–1, Santa Clara, CA, 2021. Intel Corporation.

[32] Intel Corporation. XBEGIN. In *Intel® 64 and IA-32 Architectures Software Developer's Manual Volumes 2A, 2B, 2C, and 2D: Instruction Set Reference, A-Z*, pages 5–611, Santa Clara, CA, 2021. Intel Corporation.

[33] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote flash ≈ local flash. In *ASPLOS*, pages 345–359, 2017.

[34] P. Kumar and H. H. Huang. GraphOne: A data store for real-time analytics on evolving graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 249–263, Boston, MA, Feb. 2019. USENIX Association.

[35] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[36] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, pages 317–330, 2019.

[37] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[38] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, A. Klimovic, A. Slowey, and A. Rowstron. Understanding rack-scale disaggregated storage. In *HotStorage*, 2017.

[39] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *ASPLOS*, pages 574–587, 2023.

[40] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. First-generation memory disaggregation for cloud platforms, 2022.

[41] Y. Li, R. Melhem, A. Abousamra, and A. K. Jones. Compiler-assisted data distribution for chip multiprocessors. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 501–512, 2010.

[42] C. Lu, K. Ye, G. Xu, C. Xu, and T. Bai. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *Big Data*, pages 2884 – 2892, 2017.

[43] H. Ma, S. Liu, C. Wang, Y. Qiao, M. D. Bond, S. M. Blackburn, M. Kim, and G. H. Xu. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 92107, 2022.

[44] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing MapReduce for Multicore Architectures. Technical report, Massachusetts Institute of Technology, 5 2010.

[45] H. A. Maruf and M. Chowdhury. Effectively prefetching remote memory with Leap. In *USENIX ATC*, pages 843–857, 2020.

[46] H. Moein. C++ dataframe for statistical, financial, and ml analysis. https://github.com/hosseinmoein/DataFrame, 2020.

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

[48] I. NumFOCUS. Pandas. https://pandas.pydata.org/, 2022.

[49] NVDIA. Nvidia connectx infiniband adapters. https://www.nvidia.com/en-sg/networking/infiniband-adapters, 2023.

[50] Nvidia. Virtual gpu (vgpu) | nvidia. https://www.nvidia.com/en-us/data-center/virtual-solutions/.

[51] Oracle. The java virtual machine. https://www.java.com/en/download, 2023.

[52] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, pages 361–378, 2019.

[53] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, volume 32, 2019.

[54] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira. Compiler support for selective page migration in numa architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 369380, New York, NY, USA, 2014. Association for Computing Machinery.

[55] Y. Qiao, C. Wang, Z. Ruan, A. Belay, Q. Lu, Y. Zhang, M. Kim, and G. H. Xu. Hermit: Low-Latency, High-Throughput, and transparent remote memory via Feedback-Directed asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 181–198, Boston, MA, Apr. 2023. USENIX Association.

[56] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[57] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 315–332, 2020.

[58] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, page 116, USA, 2014. USENIX Association.

[59] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[60] Y. Shan, W. Lin, R. Kosta, A. Krishnamurthy, and Y. Zhang. Optimizing hardware-based network computation dags for multiple tenants with supernic. *arXiv preprint arXiv: Arxiv-2109.07744*, 2021.

[61] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. StRoM: Smart remote memory. In *EuroSys*, 2020.

[62] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: The next generation. In *EuroSys*, 2020.

[63] L. Vilanova, L. Maudlej, S. Bergman, T. Miemietz, M. Hille, N. Asmussen, M. Roitzsch, H. Härtig, and M. Silberstein. Slashing the disaggregation tax in heterogeneous data centers with fractos. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 352367, New York, NY, USA, 2022. Association for Computing Machinery.

[64] K. Vora, R. Gupta, and G. Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 237251, New York, NY, USA, 2017. Association for Computing Machinery.

[65] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 347362, 2019.

[66] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A memory-disaggregated managed runtime. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 261–280. USENIX Association, Nov. 2020.

[67] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu. MemLiner: Lining up tracing and application for a Far-Memory-Friendly runtime. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 35–53, 2022.

[68] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *NSDI*, 2023.

[69] C. Wang, Y. Shan, P. Zuo, and H. Cui. Reinvent cloud software stacks for resource disaggregation. *Journal of Computer Science and Technology*, 38(5):949–969, 2023.

[70] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen. Exploiting program semantics to place data in hybrid memory. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 163–173, 2015.

[71] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 609621, New York, NY, USA, 2022. Association for Computing Machinery.

[72] WMT. Statistical and neural machine translation. https://statmt.org, 2011.

[73] J. Yang and J. Leskovec. Friendster social network and ground-truth communities. https://snap.stanford.edu/data/com-Friendster.html, 2012.

[74] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, 2020.

[75] H. Yu, A. M. Peters, A. Akshintala, and C. J. Rossbach. Automatic virtualization of accelerators. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 5865, New York, NY, USA, 2019. Association for Computing Machinery.

[76] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. HotCloud, page 10, Berkeley, CA, USA, 2010.

[77] M. Zhang, T. Ma, J. Hua, Z. Liu, K. Chen, N. Ding, F. Du, J. Jiang, T. Ma, and Y. Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 658–674, 2023.

[78] W. Zhang, S. Rajasekaran, S. Duan, T. Wood, and M. Zhuy. Minimizing interference and maximizing progress for hadoop virtual machines. *SIGMETRICS Perform. Eval. Rev.*, 42(4):62–71, 2015.