Systems for Memory Disaggregation: Challenges & Opportunities

Anil Yelam
ayelam@ucsd.edu

ABSTRACT

Memory disaggregation addresses memory imbalance in a cluster by decoupling CPU and memory allocations of applications while also increasing the effective memory capacity for (memory-intensive) applications beyond the local memory limit imposed by traditional fixed-capacity servers. As the network speeds in the tightly-knit environments like modern datacenters inch closer to the DRAM speeds, there has been a recent proliferation of work in this space ranging from software solutions that pool memory of traditional servers for the shared use of the cluster to systems targeting the memory disaggregation in the hardware. In this report, we look at some of these recent memory disaggregation systems and study the important factors that guide their design, such as the interface through which the memory is exposed to the application, the runtime implementation and relevant optimizations to retain the near-native application performance, various approaches to the cluster memory management to maximize memory utilization, etc. and analyze the associated trade-offs. We conclude with a discussion on some open questions and potential future directions that can render disaggregation more amenable for adoption.

1 INTRODUCTION

With the tremendous growth of computing in the past two decades, applications have become both data-intensive and latency-sensitive, which gave rise to in-memory computing in lieu of going to the disk. This led to memory-intensive applications whose memory needs on a server outweigh the processor needs, introducing a skew in resource usage. However, traditional servers come with fixed processor and memory resources that does not allow dynamically resizing memory. This was generally solved by swapping to disk but disk speeds were really slow compared to memory affecting performance. At the same time, the diversification of computing usecases introduced a high heterogeneity of applications (e.g., cloud computing) with varying memory needs in proportion to the CPU, leaving some of the traditional servers in a data center with underutilized memory and others with not enough; the result being inefficient memory utilization in the cluster and hence, increased cost of ownership. Decoupling memory would allow applications to be more elastic in their memory usage and improve the memory utilization of the cluster at the same time. Memory disaggregation involves such (logical or physical) decoupling of memory resources in a cluster from other (processor) resources.

One way to alleviate memory pressure is to scale out and build a distributed application that runs on multiple nodes and adjust itself to the memory restrictions on the individual nodes. Indeed, there are many platforms that provide distributed memory management for such applications like distributed key-value stores [15, 20, 23, 25], distributed shared memory (DSM) systems [8, 10, 16, 18], etc. These systems provide a globally accessible interface for all servers where the focus is on providing fine-grained memory sharing and a reasonable consistency model for a distributed application. An alternative is to extend the private memory space of (single-node) applications a la remote swapping systems [11, 12] that transparently swap application pages to remote memory, without any notion of sharing across servers (i.e., their memory consistency model stops with cache coherence protocols on a single server). In this report, we focus on the latter kind where the stress is more on perfomant remote access mechanisms and efficient memory management for individual applications and less on memory sharing and consistency across servers.

As the networks become faster and technologies such as RDMA [10, 32] arrive to commodity clusters, the remote access latencies are inching closer to native DRAM latencies (which, on the contrary, are nearing saturation) [3], making remote memory more accessible for applications performance-wise [13]. Consequently, there has been a renewed interest in the last half-a-decade in building remote memory systems [4, 6, 7, 9, 14, 17, 26, 27, 30, 31]. Traditional way of memory disaggregation is to pool/track unused memory across the cluster in software and use it to complement memory on the memory-hungry servers. This is still popular, with the work on remote swapping systems continuing to this day [6, 7, 14, 17]. The other, more recent approach is to disaggregate the memory in hardware and where all the memory is decoupled from compute and is made available to the compute nodes via the network [19, 22, 27]. We use the terms *remote* or *disaggregated* memory synonymously to refer to all the memory available for shared usage of the cluster whether it is pooled in software or hardware.

In both cases, building a system that exposes and manages such disaggregated memory face similar design challenges. First, the system should decide on the right interface to expose this memory; for example, to either be transparent and avoid any application changes, or to be more expressive and provide richer functionality and exploit application information/semantics for performance. Moreover, remote access latencies are still an order-of-magnitude worse than local, so the system should implement performance optimizations like caching or at the least, enable applications to implement such optimizations to hide the impact of remote latencies. While providing reasonable programming model and minimal performance degradation for a wide range of applications, it should also work towards efficiently managing the cluster memory behind the scenes and maintain good memory utilization.

In this report, we explore in detail, the above design challenges of building a system for disaggregated memory, in addition to some more that is expected of a holistic system e.g., fault tolerance, reliability, security and isolation, etc. through the lens of recent disaggregated/-far memory systems. Through this analysis, we hope to highlight the trade-offs involved with various design considerations. Finally, we conclude with a discussion on some remaining challenges and future opportunities that can help make disaggregation more amenable for adoption.

2 MEMORY DISAGGREGATION

Memory disaggregation aims to decouple the available compute and memory resources in the cluster and allow for independent allocations of these resources regardless of where a job is placed. This means, the OS/runtime that's running the job should provide a platform to expose/give access to potentially all the memory available in the cluster. Ideally, it should hide the complexity of setting up and accessing remote memory (e.g., RDMA connection setup) and expose an easy-to-use interface for working with remote memory. At the same time, it should trade-off the properties of the interface with decent performance guarantees and other requirements from the system like resource sharing and isolation across applications. At a high level, the platform is a distributed system consisting of a client-side (compute-side) components (a runtime that exposes the memory interface and acts as an agent on each compute node), the server-side (memory) components (to manage memory on each memory server) and an interconnect over which these components interact to provide an abstraction for shared cluster memory. It may optionally include other cluster resources like centralized managed for global memory/metadata management and failure handling.

Target Architecture. Proposed solutions for memory disaggregation target two different kind of cluster/memory architectures based on existing technologies or technologies that are expected to be available in the near future (shown in Figure 1).

1. Software-disaggregated. Some systems [4, 6, 11, 12, 14, 17] target the traditional homogeneous datacenters with monolithic servers as the basic deployment unit, connected to each other by low-latency network interconnects like Infiniband or RoCE. Each unit hosts both compute and memory resources and the software provides an interface to remote memory on other nodes. Local memory is prioritized for local jobs and unutilized memory on all the nodes can be pooled and presented to the cluster as remote/disaggregated memory, which could be static or vary in capacity over time.

2. Hardware-disaggregated. Other systems [7, 9, 26, 27, 31] target a hardware disaggregated architecture where (most of the) memory nodes are detached from the compute nodes and made available through the network. The memory node can be a traditional monolithic server with limited compute and stuffed with DRAM [7]

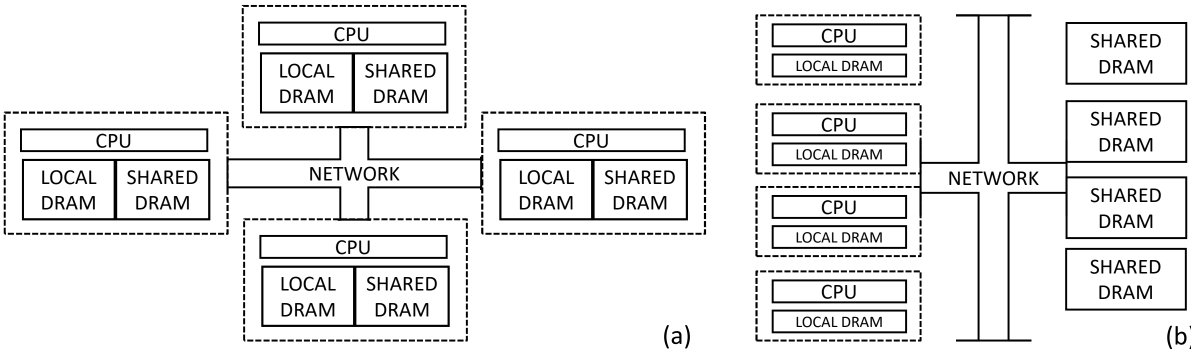

Figure 1: Shows (a) software-disaggregated architecture where disaggregated memory is pooled from traditional servers as opposed to the (b) hardware-disaggregated design where most memory is decoupled in hardware.

or each DRAM unit itself directly-attached to a memory controller and network interface [27]. Even in a purely disaggregated setup, however, it is generally assumed that each compute node has a small amount of local memory and vice versa. [9, 27]

In both architectures, compute servers use the local memory to run the OS and other runtime essentials for exposing remote memory, and only use remote memory for the applications. There are many reasons for this choice. First, without local DRAM, all the memory accesses would be remote and the memory controller should possess the knowledge and capability to fetch remote memory directly without any help from software; such complex “control path" knowledge would need either a “smart" memory controller (e.g., RMC in soNUMA [22]) or some other smart hardware (e.g., ccFPGA in Kona [9]) next to it. Even these solutions do not put OS on remote memory and and maintain local DRAM to exploit cache locality as remote accesses are still worse than local.

3 DESIGN FACTORS

We look at various considerations that guide the design of a disaggregated system and for each of them, we discuss why it matters, how previous systems have (not) treated it, are there going to be trade-offs with others, etc.

3.1 Programming Model

A key question for a disaggregated system is how it chooses to expose the memory (both local and remote) of the cluster to the applications running on it. Below, we talk about various aspects of the interface while referring to various interfaces proposed in previous systems (also listed in Table 1).

3.1.1 Transparency. Does the app (need to) know whether an access is local or remote? Does it require a rewrite of applications or can existing applications port to it with minimal or no effort?

Virtual memory-based transparency. In a traditional (x86) server, access to local memory is provided through the virtual memory abstraction using the load/store-style instructions on virtual addresses whose translation is handled by hardware (TLB, MMU) and managed by the OS. The same virtual memory abstraction can be extended to remote memory as well. The abstraction allows the actual pages to be backed my any device (disk, remote memory, files, etc.) from where they can be fetched when accessed by hooking into the page fault handler. The main benefit is complete backwards-compatibility and language-agnosticism where all the applications targeting x86 hardware can utilize remote memory without a single line of code change. The flip side is that this interface is rigid (it cannot be extended like a regular API) and kernel-based, and so doesn’t allow any app-specific information to percolate into the runtime limiting its performance optimizations, as we will see in section 3.2;

Due to its complete transparency, it has been the go-to interface for all the *remote paging* systems from the old [11, 12] that continue to this day [6, 7, 14, 17].

Interface Type (Implementation)	System	Transparent	General	What can be Remote	Sharing support
Virtual Memory (Traditional Paging)	Infiniswap [14] (2017)	Yes	Yes	All	No
	zSwap [17] (2019)	Yes	Yes	All	No
	Leap [6] (2019)	Yes	Yes	All	No
	Fastswap [7] (2020)	Yes	Yes	All	No
Virtual Memory (New Hardware)	LegoOS [27] (2018)	Yes	Yes	All	No
	Kona [9] (2021)	Yes	Yes	Heap	No
Language-based (User space)	AIFM [26] (2020)	Yes	No	Portion	No
	Semeru [31] (2020)	Yes	No	(Java) Heap	No
Custom API (Kernel-based)	Remote Regions [4] (2020)	No	Yes	Portion	Basic
	LITE MRs [30] (2017)	No	Yes	Portion	Basic

Table 1: Various interfaces for remote memory adopted in some recent systems

These systems target traditional servers (software-disaggregation) and provide remote memory as a swap space by hooking into the virtual memory manager in the kernel. In the hardware-disaggregated setting, LegoOS [27], a operating system designed for this setting, provides a similar interface for disaggregated memory. LegoOS models local DRAM as next level *virtually-indexed* cache and moves address translation hardware (TLB, paging hardware, etc.) to the memory nodes. Similar to page-faults, cache misses are handled in software when the remote pages are fetched into local DRAM. This interface also allows the whole application memory (code, stack and heap) to be in remote memory as all of it can be transparently paged out.

Kona [9] is a recent work that proposes a hardware-based implementation for this interface that avoids software overhead in handling page faults/cache misses. It proposes hardware primitives that assist with trap remote memory accesses at the memory controller hardware and fetch them from the remote memory. In an example implementation, Kona uses cache-coherent FPGA that is connected to CPU using interconnects like CXL[1]. This interconnect provides the FPGA with visibility to all the memory accesses through cache coherence protocol and routes all remote accesses through this hardware, which implements the runtime. This approach however requires special hardware.

Language-based Transparency. Without special hardware support like Kona [9], only a kernel (paging) based implementation can provide the completely transparent memory interface that can be exploited by all the traditional applications without application changes. However, kernel-based implementations suffer from incomplete information on memory access patterns and data amplification overheads because the data tracking and movement granularity (fetching remote data) is restricted by the virtual memory system i.e., the kernel must fetch the entire page (4 KB) just to access even a small object. And the simplicity of the virtual memory interface (malloc and load/store) also means that it makes it harder to provide application-specific semantics or hints to the runtime for optimized implementation. For example, data structures designed with far memory awareness might perform a lot better than traditional ones running on a memory-transparent interface [5].

To imbibe application semantics into the runtime, systems like AIFM (Application-Integrated Far Memory) [26] and Semeru [31] opt for implementing their runtime in Userspace and bypass the kernel. Since native addresses can only point to local memory, another level of indirection is needed to address memory in order to hide remote memory and retain some transparency. AIFM provides such indirection using C++ smart pointers while Semeru uses Java virtual addresses. This indirection also lets the runtime to track accesses at a much finer object granularity to perform more precise hotness tracking (leading to better cache eviction policies) and avoid data amplification. The indirection may however add some performance overhead in critical path compared to native memory accesses. Also, user space runtimes cannot place entire memory remotely (e.g., local process segments like stack and code has to be in local memory) which may offset the decoupling benefits of disaggregated memory.

Programming languages provide inbuilt implementations of common data structures like lists and hash tables either as language primitives or as a part of standard libraries. One way to take achieve app-runtime codesign while preserving transparency (i.e., no or minimal changes) for applications is to modify just these implementations under the hood to be far memory-aware. AIFM, for example, provides remoteable alternatives to standard data structures that provide access hints to the prefetcher or offload data-intensive operations like copy, aggregation, etc. to the memory server. Similarly, language runtimes can optimize their implementation around remote memory. For example, remote access latency can be hidden by maintaining lightweight threads and running other threads while some thread waits for remote data [26]. Similarly, garbage collectors in managed runtimes like Java can be optimized to target remote memory by offloading the data-intensive parts like object traversal to the memory servers [31].

No Transparency. Some interfaces provide remote memory through a custom API (usually implemented as a set of library or system calls) and are not limited by the requirement to abstract away remote memory and to be backwards-compatible. Without such a limitation, the API can choose to be very expressive. These APIs generally provide methods/calls for allocating and accessing remote memory, and in some cases, synchronization or transactional primitives for sharing memory across machines. Performance optimizations like caching are generally left to the applications. With non-transparent interfaces, however, the choice of what goes in local vs remote memory is left to the application. Since local memory is inflexible, leaving this choice to applications may hurt the ability of the runtime to efficiently perform memory decoupling and other goals of memory disaggregation.

Examples for such interfaces include Remote Regions [4] and LITE Memory regions [30] which expose an expressive kernel-based remote memory API in an effort to provide a higher-level abstraction for RDMA. These systems provide a namespace for (contiguous) memory segments (of arbitrary sizes) exposed by machines across the cluster and allow client apps to bind to these segments, and perform read/write through the ioctl stubs. The expressive interface gives them the flexibility to expose various additional operations to applications like the RPC support in LITE and caching/prefetching hints in Remote Regions. These interfaces however does not take all the complexity of careful memory management and performance optimizations but onloads some to the application. Other complex systems can certainly use these interfaces as a backend (e.g., LegoOS uses LITE as the interconnect) for ease of implementation. Other (user space) examples include the read/write API with transactional semantics provided by remote memory by recent distributed computing platforms like FARM [10] and GAM [8].

3.1.2 Generality. Is the exposed interface (ABI/API) general enough for adoption across wide range of platforms/applications, or does it favor a particular app above or particular runtime below, potentially falling out of favor with new kinds of apps/hardware? For example, interfaces provided through the kernel-based runtimes (either transparent ones based on virtual memory or non-transparent ones exposed through ioctl stubs) are more general in the sense that they're available to all the applications regardless of the language they are written in. On the other hand, in addition to being language-specific, user space runtimes [26, 31] only support remote memory access/management for a single application and hence cannot co-ordinate sharing (and isolation) of available remote memory among multiple applications. Such sharing requires mediation of the kernel, at least in the control path.

3.1.3 Ease of programming. How easy is it for applications to program with this interface? When working with remote memory, this depends on the amount of the complexity of implementation that the interface and the runtime hide away from the application. Naturally, transparent interfaces are usually the easiest to program with. Of the non-transparent ones, the complexity may vary depending on how high- or low-level the abstractions they provide are. For example, as pointed out in [3], adding two variables in disaggregated memory would be a very simple operation with virtual memory interface (*c = *a + *b). With non-transparent but still high-level abstractions like LITE [30], we need to first open the remote memory as LITE region (*LT_map()*), read the variables using *LT_read()* and write the result back using *LT_write()*. Doing the same with using RDMA would be even more complex with setting up queue pairs and memory regions, and reading/writing data by posting work requests. A common but imperfect metric to compare ease of programming is the number of lines of code (LOC). For example, both LITE [30] and Remote Regions [4] show two orders of magnitude reduction in LOC compared to pure RDMA-based implementation for various applications accessing remote memory.

3.2 Application Performance

When running on a disaggregated system, we ideally expect no or minimal degradation in application performance when compared to the native performance. Depending on the application, one can look at its job completion time, throughput or tail latencies as a proxy for performance. (Although, metrics like total cost of ownership (TCO) are more holistic and account for things like memory utilization across the cluster, special hardware costs, etc. but they are harder to measure?). In this section, we look at various factors that affect the memory performance, through these metrics.

3.2.1 Caching. Remote accesses across the network today are still on the order of magnitude slower (or worse, depending on the networking stack and interconnect), so application performance would be terrible if all accesses were to be remote [13]. Fortunately, most applications exhibit spatial and temporal locality in accesses which can be exploited by caching remote data, and as such the quality of caching can greatly affect the performance. Depending on the interface exposed, the runtime may choose to do caching or leave it to the application itself. Custom interfaces can benefit from application-specific caching hints. For example, AIFM [26] allows applications to exclude specified data from cache and avoid cache pollution.

Cache Block Size. Fetching remote data in bigger chunks will help exploit spatial locality however it also runs the risk of bringing in redundant data and polluting the cache, so it needs to be properly balanced for the best cache hit ratio. Traditional virtual memory-based approaches cannot go lower than the (4KB) page sized blocks as they hook into kernel paging; While LegoOS [27] and Kona [9] escaped this fate through hardware modifications, other systems like AIFM [26] moved to user space implementations. Kona evaluates the effect of cache block size on the performance of Redis database and determines 1KB to be optimal, which is conveniently closer to the page size. However, Kona does not do any advanced prefetching (like Leap [6]) which, in combination with smaller block sizes, may perform better than exploiting crude spatial locality with bigger blocks. Block size also effects eviction policies like LRU (which most systems use) as smaller blocks mean a larger number of blocks that need to be monitored for finding eviction candidates.

Prefetching. Prefetching remote data proactively can bring in correct pages into the cache and avoid cache misses in the critical path. Runtimes can use a transparent prefetcher that identifies access patterns and predict future accesses and/or they can provide prefetch calls in the interface that applications can inject in their code. While having a prefetcher is a more general solution, it has to balance between the accuracy of predictions and the (compute and memory) resources it consumes (lower prefetching accuracy results in cache pollution and waste of cache and I/O bandwidth). Leap [6] is an advanced prefetcher for remote paging that monitors page faults and uses the faulted addresses to predict future pages. Even with coarse information like page faults, Leap was able to achieve 1.5-2x improvement for different applications. Systems like AIFM [26] and Remote Regions [4] expose prefetch API providing applications the choice of implementing custom prefetching such as the data structure-specific ones in AIFM.

Cache Size. The bigger the size of the cache, the better; but the amount of local DRAM is limited (this limit is especially strict in hardware-disaggregated architecture where the amount of local DRAM is small and fixed). Even with the best prefetching and eviction policies, the cache size has to be enough to cover a minimum portion of the working set to avoid performance degradation and, in extreme cases, thrashing. A common chart we see in the evaluation of previous systems is to show the slowdown of an application against the local memory (or, cache size) as % of either app's peak memory usage (which varies across apps) or an arbitrary local memory capacity, and compare it to other systems; the point being no one wants to pick a particular cache size but leave that option to the reader to trade it off with performance. Looking at variety of applications across papers, it seems like the performance degradation conforms to a hockey stick pattern, remaining graceful until some 25-50% of the working set [9, 13, 27] is local and degrading dramatically below that. None of these systems give an idea as to the one-size-fits-all limit for the absolute size of local DRAM though.

3.2.2 Interface Overhead. The interface itself can add some software overhead to the each memory access. Unlike virtual memory interfaces that use native load/-store, user space interfaces involve either library calls or another level of pointer indirection (e.g., Java) that may add few cycles for each operation. For example, AIFM uses C++ smart pointers and when compared to Fastswap that allows native pointers, it adds a marginal overhead that becomes evident when effects of their runtime and interconnect are minimized. (Fig 7 [26]). Similarly, Kona [9] that routes remote accesses through the cache-coherent hardware that exposes remote memory as another physical DRAM whose accesses are slower compared to regular DRAM due to limited interconnect bandwidth. Kernel-based custom interfaces like LITE [30] introduce syscall in the access path that adds significant overhead. LITE however is a low-level interface that leaves caching to the application and such accesses can be made in the cache miss path.

3.2.3 Remote Access Latency. Optimizing the actual latency in fetching remote data (i.e., the cache miss path) is important not only because caching cannot soften the performance impact if the miss latency is too high but also because it impacts tail-latencies that modern datacenter applications are sensitive to. This latency depends on both the implementation overheads on the client and the server side, and the choice of the interconnect itself. We discuss these factors below, in addition to the previous optimizations to reduce or hide this latency.

Network Interconnect. Gao et al. [13] analyzed the effect of increasing remote access latency for various applications and arrive at 4-5 μs upper bound to maintain performance. (It was, however, done on paging-based systems with default page replacement algorithms not optimized for remote memory, and hence should be taken as a rough estimate). Such low latencies are now possible in tight-knit local area networks like modern data center racks, where TCP/IP and, more recently, RDMA [32] have become standard transport options. RDMA has been the preferred target transport in most of the recent systems because it cuts down on the software stack on both sides and, more so, because it provides one-sided accesses that avoid remote CPU in critical path. For example, systems that [26, 30] explored the TCP/IP option reported up to 10 μs overhead compared to RDMA for remote operations. Technologies like Intel Omnipath [2] and CCIX [1] are expected to further slash the latency by bringing the NIC much closer to the CPU, once they are commercially available.

Minimizing/avoiding software overheads. With respect to the remote access latency, the goal for the runtime is to get keep it as close as possible to the raw network latency and minimize any software overheads. The main software overhead comes from finding space for incoming remote data and deciding which data to evict to make that space (i.e., the typical cache miss handling), before returning to the application.

Paging-based systems started with conventional disk-paging subsystem and over time improved it to target remote memory where paging is more frequent and operates in microsecond timescales instead of milliseconds [20]. A common optimization is to decouple allocation and eviction by maintaining a free list for newly allocated or fetched-in data and moving eviction to the background and off the critical path [6, 20]. For example, Fastswap [7] offloads memory reclamation to a dedicated CPU core so that the application CPU can return to user space and continue its execution. Another (de-facto) optimization is to allocate local pages for newly allocated memory and keep it local until evicted (i.e., delayed write-back). When bringing additional data along with the required data (either because of prefetching or high data granularity to exploit locality), it is recommended to fetch the additional data separately outside the critical path [7] to avoid head-of-the-line blocking.

Fundamentally though, as long as cache misses (page faults) are handled in software (the only available option with traditional hardware; although LegoOS, with hardware modifications, still takes the software route due to miss path complexity), the software miss path will inevitably involve overheads like context-switching, CPU cache pollution, etc. that add to the latency. Kona [9], for this reason, proposes new hardware primitives to offload this path to avoid above issues and also benefit from the resulting hardware speedup. The flip side, of course, is the complexity of implementing part of the runtime and networking stack in hardware.

Hiding the latency. From a throughput (CPU utilization) standpoint, latency can be hidden by switching out the current thread and running a different thread while the remote data is fetched. Kernel thread context switches are on the order of microseconds and may not result in much savings but user space runtimes with light-weight threads can use this approach, like AIFM did. [26].

3.2.4 Reducing remote accesses with RPCs. Some access patterns like traversals may be ill-suited for remote fetching and cause too many cache misses no matter how good caching/prefetching is, and only viable option might be executing those operations closer to the memory using remote procedures (RPCs). To support this, systems may allow applications to register/invoke methods on the remote node, such as function shipping in FaRM [10], AIFM Remote Devices [26], LITE RPC [30], etc. AIFM gets most of its performance benefits by sending such memory intensive operations of its data structures to the memory server. Semeru [31] does the same for garbage collecting the disaggregated Java Heap. Enabling this, however, depends on the assumed capabilities of the memory server.

3.3 Memory Management

Similar to virtual memory interface on traditional servers, a system for system for disaggregated memory needs to pool the memory on multiple servers/nodes and present a unified interface for allocating and accessing this memory by hiding away the underlying physical locations. It should be further responsible for tracking available memory across the nodes, map it to an application on allocation and fetch the data when required. Most systems [4, 9, 27] employ a global memory manager that maintains this metadata and works with agents/daemons on the (memory) nodes to find space for new allocations. Compute nodes query the manager for new allocations and cache the mappings to directly go to the relevant memory node during access time. Under the hood, the system should aim for better memory efficiency while providing good data path performance and scalability. Below, we discuss few factors that affect these goals.

3.3.1 Memory backing or complementing. In a hardware disaggregated setting, all application memory is allocated in the disaggregated memory and local DRAM is very small and only used as cache i.e, all the app memory is backed by disaggregated memory. In a software-disaggregated setup, a similar approach can be taken by reserving a small amount of local DRAM on each node for local workspace and the rest is pooled from which memory is transparently allocated to all the applications. However, this fails to exploit local affinity in such a non-uniform memory access setting so all systems proposed for traditional setting prioritize local allocations and only expose unutilized memory for shared cluster use. Such disaggregated pool can then be used for complementing (not backing) the local memory either by transparently expanding local address space (i.e., remote paging systems) or through other interfaces like mmapped files [4].

3.3.2 Memory tracking granularity. Just like paged memory, memory allocation and mapping is done is fixed-size units (slabs). Although extending page size (4KB) to remote memory would be a natural choice, it adds a significant 0.2% space overhead for maintaining mapping metadata (e.g., 2 GB for a 1 TB region[4]), so most systems end up using a much bigger slab size (128B [4] segments or "multiple" pages [9, 14]). Local memory allocators can then manage these slabs for fine-grained allocations. However, bigger sizes may cause internal fragmentation or require contiguous segments at memory servers and hurt memory efficiency. None of the works, however, evaluate this aspect and slab sizes were arbitrary.

3.3.3 Balancing memory usage across nodes. Remote memory used by an application is better if uniformly distributed across multiple memory nodes both to balance the access load and to minimize the performance impact in case of remote node failures. With a global memory manager, we can maintain the available memory across the memory nodes and properly direct memory allocation requests to both distribute memory footprint of application as well as balance overall memory usage across the memory nodes [4, 9, 27]. However, both the overhead of constantly communicating with memory nodes and the centralized manager may present a scalability bottleneck. Infiniswap [14] takes a decentralized approach where nodes requesting memory choose randomly from the list of available nodes. To help reduce imbalance, it opts for power of two choices [21] where instead of randomly picking one, two nodes are picked randomly and the one with the most available memory is chosen. However, it is unclear if this approach sustains the balance with increasing scale or over time.

3.3.4 Memory reclamation on the server. As mentioned before, software-disaggregated systems need to balance available memory on each node between local and global memory (with priority for local use), and hence should should be amenable to reclamation to make space for expanding local usage. Infiniswap [14], which uses remote memory as swap space, writes swapped out pages to both remote memory and disk, and hence can afford to drop the remote data during reclamation on a remote node. Reclamation is less of an issue in the hardware disaggregated setting but they may still need to swap out some memory to disk under extreme cluster memory pressure. An interesting, rather unexplored, question then is which memory should be picked to drop/evict to minimize performance degradation. Traditional systems use LRU lists based on page access information to swap out colder pages however such hotness information is either not available or, at best, distributed across compute nodes in a disaggregated system. Infiniswap [14] again uses power of choices to make an informed guess (where the memory node queries a random subset of compute nodes for this hotness information and drop relatively colder pages) rather than randomly dropping any arbitrary pages.

3.3.5 Summary. Memory efficiency in general is not very well evaluated in any of the systems so far (only Infiniswap [14] had a chart showing cluster memory utilization over time), perhaps because of the focus on application performance which is the prime roadblock to feasible disaggregation. Userspace runtimes completely punt on the memory management aspect and assume that required memory is pre-allocated [31] or limit themselves to working with a single memory server [26]. While they spotlighted the performance advantages of app integration, these systems most certainly need further work on the memory management side to be considered feasible for adoption.

3.4 Other Considerations

There are other considerations that received less focus so far and were often overlooked in previous systems but needs to be worked on for adoption in the wild. As work on performance tuning saturates, we expect there to be more focus on these topics in future.

3.4.1 Fault Tolerance & Reliability. By distributing the memory across multiple servers/nodes, memory disaggregation expands the fault domain of an application and makes it more prone to failures. Our system should account for this; we only need to worry about remote node failures as local node failures occur in traditional setup too and the goal is for our system to be as fault-tolerant as the traditional setup.

One option is in-memory replication [6, 9]. This, however, would consume at least twice as much memory, wasting precious DRAM and may prove too high of a cost. LegoOS [27] only maintains the log on secondary replica to minimize the memory overhead risking higher recovery time after a failure. Another alternative is to write the remote pages/data to persistent storage (disk) in the background along with storing it in remote memory [14]. However, remote write load may be too high for disk to handle which may result in disk queue build-ups. Recovering from failure would also be slower but this may be tolerable if the failures are rare. In both cases though, replication/writing to disk only happen during evictions which are generally off the critical path, and hence won't directly affect application performance. It is not clear, however, whether we can get away without any of these and just fail the applications using a memory node when it crashes.

Since network is involved now, transient network failures or congestion may affect remote accesses. Normally this only degrades performance due to timeouts/retries during page faults/misses but does not affect fault tolerance. In Kona [9] however, remote accesses are served using special hardware through cache-coherent protocol that is sensitive to memory access latencies and may end up crashing the application.

3.4.2 Network Efficiency. Network bandwidth may become a limiting factor and so better network efficiency would keep the network less congested and latencies low. Amount of networked data is primarily affected by quality of caching (cache misses bring in the data) and the evictions (writing the dirty data back). Bigger cache blocks, like the page size in most systems, will cause data amplification (by bringing in redundant data) and hurt network efficiency, especially during the write-backs where dirty data is usually only a small fraction of the block size. Paging approaches cannot track dirty data on finer granularity but Kona [9] (with special hardware) and AIFM [26] with userspace runtimes can, and avoid I/O amplification by just writing this data. For example, Kona uses a log to track multiple dirty cache lines and sends it to the memory node in batches.

3.4.3 Security & Isolation. Kernel is (has to be) trusted, and as long as the runtime (and agents on all the components) work in kernel space, remote memory can be indirectly provided to application where these indirection mapping/translation is controlled by the kernel, providing a security isolation similar to that provided by the traditional virtual memory. All kernel-based systems provide this support to enable safe access to remote memory in presence of multiple applications. User space runtimes cannot implement isolation and support sharing between applications as they are restricted to a single application. Such runtimes may need to bank on kernel-based, lower-level interfaces like Remote Regions [4] or LITE [30] for proper isolation in presence of multiple applications using such runtimes. Even if an attacker does not control these components, disaggregated memory increases the attack surface and may enable side-channel attacks such as Pythia [29], so more work on security is certainly needed.

4 DISCUSSION

In this section, we first discuss some open questions regarding the design that haven't been categorically answered so far. We then talk about a few potential directions for future exploration.

4.1 Open Questions

Transparency vs. App Integration Transparency is important for adoption so, as we've seen, most previous systems opt for it. However, as language-based runtimes have shown, exploiting application hints/semantics for optimizing runtime can be a key factor in cutting down the performance impact. Perhaps there is a middle-ground to be found here where a system provides both options and the choice is left to the application to either run without changes or provide hints for better performance. For example, a kernel-based virtual memory interface that also exposes a custom API through ioctl (like LITE [30]) to, for example, supply application-specific prefetching or call an RPC.

The right amount of memory sharing (and consistency). Systems so far have either completely avoided any explicit memory sharing across servers or provided only coarse-grained sharing with no consistency guarantees. If sharing is to be supported for transparent virtual memory interface, it should provide same semantics as current cache coherence protocols on the entire address space but providing such strong consistency model across the network is still infeasible (even the recent RDMA-based DSMs [8, 10] do not attempt it); none of the major (transparent) disaggregation systems [14, 27] provided this support, perhaps for this reason. Non-transparent ones like Remote Regions [4] and LITE [30] provide support for mapping the same region in multiple servers along with basic synchronization support like barriers and mutexes for synchronizing their accesses, but no consistency on reads/writes. Sharing generally disallows caching and delayed write-backs, both of which are critical to making performance of remote memory feasible for applications, as we have seen in section 3.2. Aguilera et al. [3] mooted the idea of non-simultaneous sharing where at any time, memory is accessed exclusively by a single host, but across time it can be accessed by many; which is enough for some common distributed workloads like MapReduce.

The extent of memory-side compute. Most previous systems differ significantly on the amount of remote/near-memory compute capabilities they expect of the memory server. At the bare minimum, the memory server should be able to serve remote memory allocations and accesses while managing its local memory. On top of that, some systems offload other runtime tasks such as address mapping, replication and persisting to disk [27], and even garbage collection [31] to the memory servers. Others [26, 30] go even further and introduce RPCs that applications can register/run on the memory server, which can be of arbitrary complexity but help with performance. However, this may be a slippery slope as it adds complexity to the memory server that we do not want, at least in the hardware disaggregated setting where there is limited compute on memory nodes. Even in the traditional setup, we are still looking to decouple CPU from memory, and increasing compute complexity on the remote memory side does not help with disaggregation and only complicates resource accounting. Perhaps a middle ground is to find a set of hardware-friendly primitives (as in [5]) for common remote side operations and expose only these set as RPCs through platforms like Storm [24] and Strom [28] that enable implementing these RPCs on the remote NICs.

The right benchmark to evaluate with. A disaggregated system is expected to be general and support a wide variety of applications, not just interface-wise but also w.r.t. the performance impact. As many recent systems show in their evaluation [7, 13, 27], performance degradation can vary dramatically across applications depending on the amount of local memory. Similarly, different applications react differently to perfomance optimizations like caching and prefetching, based on their memory access and locality patterns. Given this heterogeneity, the benchmark for evaluating a diaggregated memory system should include a variety of applications ranging from compute to memory intensive ones. All recent systems, however, chose their own set of custom benchmarking applications and very few of them intersect making the comparison across the spectrum infeasible as the chosen benchmark may favor their own systems. It is, therefore, preferable to have a standard benchmark.

4.2 More Opportunities for Future

Disaggregated Memory for VMs and beyond. Most modern workloads run on some virtualization platform or the other (VMs, Containers, Lambdas, etc.) [3], so there is a need to extend disaggregated memory to such platforms for wider adoption. The hypervisor can take the place of the kernel in implementing runtime and managing/providing the diaggregated memory. However, VMs, unlike processes, come with strict SLOs and with an expectation of performance isolation, so the runtime needs to account for this while managing memory (e.g., proper memory accouting, maintaining separate caches, etc.). Also, the gap between the application and the runtime is further widened making it harder to get app-specific information.

Job scheduling on disaggregated systems. Ideally, job scheduling based on its CPU and memory requirements should be simple on a disaggregated system because as long as the cluster has enough memory for a job, it can be placed anywhere. However, non-uniform memory access between local and remote memory means that cluster throughput would be higher if total number of remote accesses, which can depend on job placement, is minimized. For example, given similar compute requirements, jobs with large working memory are better paired with the ones with smaller working sets on the individual compute nodes to balance out contention for the local DRAM (cache). Similarly, different jobs react differently to reduced local memory (cache) size which is another aspect on which jobs can be balanced to prioritize the sensitive ones. This line of work can build on far-memory aware schedulers like Fastswap [7].

Exploiting other new trends in the datacenters. With datacenters increasingly employing heterogeneous hardware, disaggregated memory should be made accessible to these devices as well. This raises the question of the best interface for accessing memory from such custom devices like GPUs or FPGAs, and how feasible it is for implementation on these devices. For example, traditional paging-based interfaces cannot be implemented on custom hardware that lacks the TLB and MMU hardware (LegoOS [27] moves the translation hardware to memory nodes making it friendlier for such hardware). Another related trend is the adoption of programmable networking hardware like NICs and switches. Switches have been used to accelerate applications through in-switch caching, load balancing, etc. Similar acceleration opporunities may be found in case of diaggregated memory systems for global tasks like memory management and load balancing in the critical path.

REFERENCES

[1] 2021. CCIX: cache coherent interconnect for accelerators. https://www.ccixconsortium.com/.

[2] 2021. Intel OmniPath Architecture. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-driving-exascale-computing.html.

[3] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote memory in the age of fast networks. *SoCC 2017 - Proceedings of the 2017 Symposium on Cloud Computing* (2017), 121–127. https://doi.org/10.1145/3127479.3131612

[4] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2020. Remote regions: A simple abstraction for remote memory. *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018* (2020), 775–787.

[5] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing Far Memory Data Structures: Think Outside the Box. *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019* (2019), 120–126. https://doi.org/10.1145/3317550.3321433

[6] Hassan Al Maruf and Mosharaf Chowdhury. 2019. *Effectively Prefetching Remote Memory with Leap*. https://www.usenix.org/conference/atc20/presentation/al-maruf

[7] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the 15th European Conference on Computer Systems, EuroSys 2020*. ACM, 16. https://doi.org/10.1145/3342195.3387522

[8] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chenz, Beng Chin Ooi, Kian Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient distributed memory management with RDMA and caching. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1604–1617. https://doi.org/10.14778/3236187.3236209

[9] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *ASPLOS*. 79–92. https://doi.org/10.1145/3445814.3446713

[10] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014* (2014), 401–414.

[11] Sandhya Dwarkadas, Nikolaos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil, and Robert Stets. 1999. Cashmere-VLM: Remote memory paging for software distributed shared memory. *Proceedings of the International Parallel Processing Symposium, IPPS* (1999), 153–159. https://doi.org/10.1109/ipps.1999.760451

[12] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. 1995. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 201–212. https://doi.org/10.1145/224056.224072

[13] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 249–264. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao

[14] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. *Efficient memory disaggregation with Infiniswap*. 649–667 pages. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[15] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2015. Using RDMA efficiently for key-value services. *Computer Communication Review* 44, 4 (2015), 295–306. https://doi.org/10.1145/2619239.2626299

[16] Pete Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. (1994).

[17] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS* (2019), 317–330. https://doi.org/10.1145/3297858.3304053

[18] Kai Li and Paul Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems (TOCS)* 7, 4 (1989), 321–359. https://doi.org/10.1145/75104.75105

[19] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings - International Symposium on Computer Architecture*. 267–278. https://doi.org/10.1145/1555754.1555789

[20] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. *System-level implications of disaggregated memory*. Technical Report. 1–12 pages. https://doi.org/10.1109/hpca.2012.6168955

[21] Michael David Mitzenmacher and Alistair Sinclair. 1996. *The Power of Two Choices in Randomized Load Balancing*. Ph.D. Dissertation. AAI9723118.

[22] Stanko Novaković, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-Out NUMA. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS* (2014), 3–17. https://doi.org/10.1145/2541940.2541965

[23] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2016. The case for rackout: Scalable data serving using rack-scale systems. In *Proceedings of the 7th ACM Symposium on Cloud Computing, SoCC 2016*. 182–195. https://doi.org/10.1145/2987550.2987577

[24] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. 2019. StoRM: A fast transactional dataplane for remote data structures. *SYSTOR 2019 - Proceedings of the 12th ACM International Systems and Storage Conference* (2019), 97–108. https://doi.org/10.1145/3319647.3325827 arXiv:1902.02411

[25] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M Rumble, Eric Stratmann, and Ryan Stutsman. 2009. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *Stanfordedu* 43 (2009), 1–14. http://www.stanford.edu/$\sim$ouster/cgi-bin/papers/ramcloud.pdf

[26] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, application-integrated far memory. *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020* (2020), 315–332.

[27] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2007. Legoos: A disseminated, distributed OS for hardware resource disaggregation. *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018* (2007), 69–87. https://www.usenix.org/conference/osdi18/presentation/shan

[28] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: Smart remote memory. *Proceedings of the 15th European Conference on Computer Systems, EuroSys 2020* (2020). https://doi.org/10.1145/3342195.3387519

[29] Shin Yeh Tsai, Mathias Payer, and Yiying Zhang. 2019. *Pythia: Remote oracles for the masses.* 693–710 pages. https://www.usenix.org/conference/usenixsecurity19/presentation/tsai

[30] Shin Yeh Tsai and Yiying Zhang. 2017. LITE Kernel RDMA Support for Datacenter Applications. *SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles* (2017), 306–324. https://doi.org/10.1145/3132747.3132762

[31] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. *Semeru: A memory-disaggregated managed runtime.* 261–280 pages. https://www.usenix.org/conference/osdi20/presentation/wang

[32] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. *Computer Communication Review* 45, 4 (2015), 523–536. https://doi.org/10.1145/2785956.2787484