How To Build Your Disaggregated Memory System

Anil Yelam
ayelam@ucsd.edu

ABSTRACT

1 INTRODUCTION

With the tremendous growth of computing in the past two decades, applications has 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. Traditionally, this was 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.

Meta ►Scoping◄ An obvious way to alleviate memory pressure is to build a distributed application that runs on multiple nodes and adjust itself to the memory restrictions on the individual nodes. Indeed, there is a lot of work here that focus on building performant distributed applications, like the in-memory data stores (e.g., RAMCloud, RDMA-based KVS papers, etc. TODO ►cite some◄). However, these solutions are specific to those applications and do not generalize. However, we are looking for general platforms (rather than individual applications) that provide (somewhat) generic, low-level interfaces targeting a wide spectrum of applications and allow them to efficiently utilize the entire memory of the cluster. Some of the earlier systems that provided such mechanisms include the remote swapping systems [11, 12] that transparently moved around the pages of virtual memory subsystem in the cluster, and distributed shared memory systems [15, 17] that did the same behind a global virtual address space. Anil ►Remote memory draws from this lesson and avoids the hard problem of sharing: its goal is not to implement shared memory. In fact, remote memory need not be shared at all: it can be used to store private pages to extend local memory or to safeguard data remotely. Or it can provide a different form of sharing: we envision non-simultaneous sharing of remote memory, whereby a host stops using the data before another node starts (e.g., in different phases of map-reduce).◄

Anil ►There are many ways for a machine to store data remotely, such as using key-value storage systems (e.g., [16, 17, 28, 36]), tuples [14], distributed objects [45], files, database systems, and RDMA [4, 5]. While effective, these abstractions are fundamen- tally different from the abstraction of memory accessed via◄ 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 work on remote swapping systems continuing to this day [6, 7, 14, 16]. The other, more recent-style is to disaggregate the memory in hardware and is available to all the compute nodes through the network [25]. In both cases, building a system that exposes and manages such disaggregated memory face very 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 to provide

richer functionality and allow app optimizations. Remote access latencies are still an order-of-magnitude worse than local, so the system should decide on performance optimizations like caching or at least, enable applications to implement their optimizations. While providing reasonable programming model and performance for a wide set of applications, it should also work towards efficiently managing the cluster memory behind the scenes and maintain good utilization - our goal in the first place.

In this report, we explore in detail, the above design challenges of building a system for disaggregated memory, and 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. [4, 6, 7, 9, 14, 16, 24, 25, 28, 29]. Through this analysis, we hope to highlight the tradeoffs involved with various design considerations and challenges left for widespread adoption. We end with a discussion on TODO ►◄.

2 BACKGROUND

Historical work. Very early work on improving cluster memory utilization [11, 12, 15, 17] sticked to traditional server architectures and focused on software-based memory pooling across the cluster and exposing it to applications in (mainly) two different flavors. Distributed shared memory (DSM) systems [15, 17] provide a global shared address space for writing distributed applications, where the address space is globally accessible from all the servers. These systems can then transparently back subsets of address space (at different granularity) with physical memory from different servers across the cluster, and therefore have the flexibility of optimizing cluster memory utilization. Other systems like GMS [11, 12] just focus on extending the address space of individual applications without any notion of sharing across servers (i.e., their memory consistency model stops with cache coherence protocols on a single server). These systems merely back or complement local DRAM with unutilized memory from other servers using the paging mechanisms in the operating system, usually by providing remote memory as another block device to swap to. DSM systems provide shared memory (along with some consistency model) which makes distributed programming easier but they may require applications be rewritten using their memory model. Conversely, remote paging systems aim to be more backwards-compatible and leave the complexity of building distributed applications (if needed) to the higher layers. These early systems however hasn't seen adoption (perhaps?) because remote memory latencies were still far higher for tenable application performance. (DSM though had even more challenges due to overheads in ensuring consistency that get amplified with slow networks).

Recent proliferation in this space. With the advent of faster networks and technologies such as RDMA [10] to commodity clusters, there has been a renewed interest in building such systems. As remote access latencies get closer to native DRAM latencies (which, on the contrary, are nearing saturation [3]), writing applications with such accesses in the critical path is looking feasible, performance-wise [13]. Consequently, there has been a lot of RDMA-inspired remote memory system building in recent years, including a renewal in DSM [8, 10, 21, 26] and Remote paging [6, 7, 14, 16, 18, 19] systems. While the remote paging systems provide native virtual memory interface (i.e., memory access through load/store ops on cached local pages from remote memory) to applications, other new interfaces for remote memory were proposed that sacrifice application transparency in favor of performance due to either the simplicity of implementation [4, 28] (light-weight runtime means less performance overhead in the critical access path) or benefits from application hints [24]; conversely, these interfaces require app modifications but enable applications to distinguish between local and remote memory accesses and be smart about it e.g., use far-memory aware implementations [5, 29].

Most systems for disaggregated memory target clusters with traditional (commodity) servers where memory is collocated with processors and there is no special hardware support. Traditional hardware cannot access remote memory directly and hence usual solutions has to proxy it through local memory by caching remote data and resorting to software to fetch remote data on a cache miss. To avoid this overhead, some systems introduce special hardware like a remote memory controller [22] or a cache-coherent FPGA [9] to plug into the local cache hierarchy and handle remote accesses through the hardware. And finally, as an alternative to traditional server model, there were proposals for

new hardware architectures where memory is decoupled and pooled at a hardware-level, and some systems targeting such disaggregation [18, 22, 25] as well, a note-worthy one of which is LegoOS [25]. In this report, we will look at many of these systems to provide an informed view of disaggregated system design.

3 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 in the cluster. 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 and queue pair management) 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) component (a runtime that exposes the memory interface and acts as an agent on each compute node), the server-side (memory) component (to manage memory on the server) and an interconnect over which these components interact to provide an abstraction for shared cluster memory. It may optionally include other cluster resources for global memory/metadata management. TODO ►figure?◄

3.1 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; we look at system design targeting both these architectures (shown in Figure 1).

Software-disaggregated. Some systems [4, 6, 11, 12, 14, 16] 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.

Hardware-disaggregated Other systems [7, 9, 24, 29], like LegoOS [25], 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] or each DRAM unit itself directly-attached to a memory controller and network interface [25]. 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, 25]

In both architectures, compute servers use local memory to run the OS and other runtime essentials for exposing remote memory[1], 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., ccF-PGA 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 an at least an order of magnitude higher than local.

Anil ►explain other benefits like fault tolerance?◄

4 DESIGN

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.

4.1 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,

[1] we use the terms remote or disaggregated memory synonymously to refer to all the memory available for shared usage of the cluster in both architectures

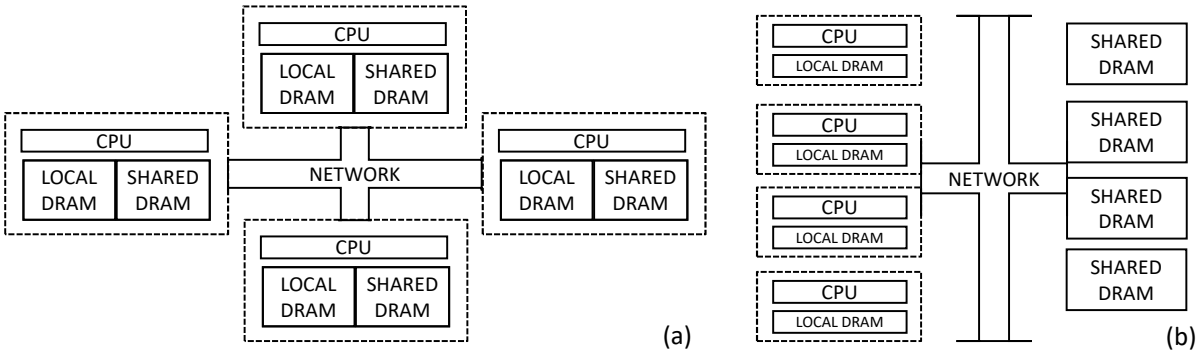

Figure 1: (too big?) Shows (a) software-disaggregated architecture where disaggregated memory is pooled from traditional servers as opposed to the (b) hardware-disaggregated design where most memory is decoupled in hardware.

System	Memory Interface	Interposition Layer	Trans-parent?	General?	What's Remote	Sharing support?
Infiniswap [14]	Virt Mem	Kernel (Paging)	Yes	Yes	All	No
zSwap [16]	Virt Mem	Kernel (Paging)	Yes	Yes	All	No
Leap [6]	Virt Mem	Kernel (Paging)	Yes	Yes	All	No
Fastswap [7]	Virt Mem	Kernel (Paging)	Yes	Yes	All	No
LegoOS [25]	Virt Mem	Kernel/Hardware	Yes	Yes	All	No
Kona [9]	Virt Mem	Hardware	Yes	Yes	Heap	No
Fastswap [7]	Virt Mem	Kernel (Paging)	Yes	Yes	All	No
AIFM [24]	C++	User space	Yes	No	Portion	No
Semeru [29]	Java	User space	Yes	No	(Java) Heap	No
Remote Regions [4]	Custom	Kernel (ioctl)	No	Yes	Portion	Basic
LITE MRs [28]	Custom	Kernel (ioctl)	No	Yes	Portion	Basic

Table 1: Interfaces for remote memory adopted in some recent systems TODO ▶is this table really adding any value?◀

we talk about various aspects of the interface, while referring to previous systems (Table 1).

4.1.1 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 and limits its performance optimizations, as we will see in section 4.2;

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, 16]. 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 [25], 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 I/O amplification and incomplete information on memory access patterns 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, similar to NUMA-aware data structures, data structures designed with far memory awareness might perform 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) [24] and Semeru [29] 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 (e.g., scalability).

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 [24]. 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 [29].

No Transparency. Some interfaces provide remote memory through a custom APIs (usually implemented as a set of library or system calls) that 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 [28] that 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 read/write through the ioctl stubs. The more expressive interface gives them the flexibility to expose various additional operations like the RPC support in LITE and caching/prefetching hints in Remote Regions. These interfaces however keep it low-level and does not take all the complexity of careful memory management and performance optimizations (next section) and limit themselves to naming and providing fast access to remote memory segments. Other complex systems can certainly use these interfaces (e.g., LegoOS uses LITE as the interconnect) for ease of implementation. Other (user space) examples include the transactional read/write semantics provided by remote memory by recent distributed computing platforms like FARM [10] and GAM [8].

4.1.2 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) cater to all the applications regardless of the language they are written in. In addition to being language-specific, user space runtimes [24, 29] 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.

4.1.3 Ease of programming. How easy is it for applications to program with this interface? When working with remote memory, this depends on how much of the complexity of implementation does 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-level or low-level the abstractions they provide are. As pointed out in [3], adding two variables in disaggregated memory would be a simple operation with virtual memory interface (*c = *a + *b). With non-transparent but still high-level abstractions like LITE [28], we need to first open/map the remote memory as LITE region, read 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 is to compare number of lines of code (LOC). For example, Both LITE [28] and Remote Regions [4] show two orders of magnitude reduction in LOC compared to RDMA-based implementation. TODO ▶figure showing a typical transparent vs non-transparent interface◀

4.1.4 RPCs. Some operations may be highly sensitive to remote accesses (like graph traversal) and can benefit immensely from having the operation run on a memory server (either in hardware or software depending on the memory server architecture). To support this, interfaces allow applications to register/invoke methods on the remote node, such as function shipping in FaRM [10], AIFM Remote Devices [24], LITE RPC [28], etc. Their scope is generally limited to simple (pre-determined) operations on few memory locations known to be on the memory server. Depending on server capabilities, they may not be entirely feasible in hardware disaggregated scenarios. Perhaps a middle ground is find a set of hardware primitives (as in [5]) for common remote side operations and expose only these set as RPCs through platforms like Storm [23] and Strom [27] that enable implementing these RPCs on the remote NICs.

4.1.5 Sharing across nodes. Does the interface allow sharing the disaggregated memory across multiple nodes? If so, what kind of consistency model does it provide/allow? What kind of synchronization primitives does it provide (basic or advanced)? 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 recent DSMs [8, 10] based on RDMA do not attempt it); none of the major (transparent) systems [14, 25] provided this support, perhaps for this reason. Non-transparent ones like Remote Regions [4] and LITE [28] 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 will see in the next section. TODO ►needs work◄.

4.2 Application Performance

When running on a disaggregated system, we ideally expect no or minimal degradation in application performance when compared to local memory 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.

4.2.1 Caching. Remote accesses across the network today are still on the order of 10-100x slower with 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 enable application-specific caching hints. For example, AIFM [24] allows applications to exclude caching specified data 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 [25] and Kona [9] escaped this fate through hardware modifications, other systems like AIFM [24] 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 [24] 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, 25]. None of these systems give an idea as to the one-size-fits-all limit for the absolute size of local DRAM though. **Anil** ▶double check these assertions. cache size is important because it gives a sense of how many/what kind of app mixture can be run on compute node without thrashing◀

4.2.2 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 [24]). 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 **Anil** ▶not sure how much◀.

Kernel- based custom interfaces like LITE [28] 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 must be made in the cache miss path.

4.2.3 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 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 [30] 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 [24, 28] 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. **Anil** ▶talk about the role of network congestion control?◀

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 [19]. 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, 19]. 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 (defacto) 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 alternative with traditional hardware; although LegoOS, with hardware modifications, still takes the software route due to miss path complexity), the software miss path will inevitably cause context-switching, CPU cache pollution, etc. that adds to the latency. Kona [9], for this reason, proposes new hardware primitives to offload this path to avoid above issues and also benefit from hardware speedup. The flip side, of course, is the complexity of implementing part of the runtime and networking stack in hardware. TODO ►refine last two sentences◄.

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. [24].

4.2.4 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, only good thing might be executing those operations closer to the memory using RPCs. AIFM gets most of its performance TODO ►how much? try to quantify it.◄ by sending such memory intensive operations of its data structures to the memory server. Semeru [29] does the same for garbage collecting the disaggregated Java Heap. This, however, feels like a slippery slope as it adds complexity to the memory server that we don't want to, at least in the hardware-disaggregated setup where there is limited compute on memory nodes. Even in the traditional setup, we are still looking to decouple CPU from memory and a compute complexity on the memory side does not help with the disaggregation. TODO ►needs work◄

4.3 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 responsible for tracking available memory across the nodes, map it to an application on allocation and fetch the data when required. With most systems [4, 9, 25], there is 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.

4.3.1 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 nodes. However, this fails to exploit local affinity in such NUMA setting so all systems proposed for this setting prioritize local allocations and only expose unutilized memory for shared cluster uses. Such disaggregated pool can then be used for complementing local memory (not backing it) either by transparently expanding local address space (i.e., remote paging systems) or through other interfaces like mmapped files [4].

4.3.2 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 size (128B [4] segments or "multiple" pages [9, 14]). Local memory allocators can them 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.

4.3.3 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, 25]. 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 [26] 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 in presence of allocations and freeing over time.

4.3.4 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. This is less of an issue in 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 to minimize performance degradation. Traditional systems use LRU lists based on 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 where memory node queries a random subset of compute nodes for this hotness information and drop relatively colder pages rather than randomly dropping any pages.

4.3.5 Summary. Memory efficiency in general is not very well evaluated in any of the systems so far (only Infiniswap [14] had a chart on cluster memory utilization), perhaps because of the focus on application performance which is the prime roadblock to feasible disaggregation. Userspace runtimes [24, 29] completely punt on the memory management aspect and assume that required memory is pre-allocated or limit themselves to working with a single memory server. While they explored the performance advantages of user space, these systems most certainly need further work on the memory management side to be considered feasible for adoption.

4.4 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.

4.4.1 Fault Tolerance & Reliability. By spreading 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 generally need to worry about remote node failures as local node failures occur in traditional setup too and we only want our system to be no less fault-tolerant than 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 [25] only maintains the log on secondary replica to minimize the memory overhead. 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 results in disk queue build-ups. Recovering from failure would also be slow but this is tolerable considering that failures are rare. In both cases, 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 though whether we

can get away without any of these and just fail the applications using a memory node when it crashes. Anil ▶LegoOS did some analysis on mean time to failure for hardware that may help here.◀

Since network is involved now, network failures may cause timeouts in 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.

4.4.2 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 [24] with userspace runtimes can, and avoid I/O amplification by just writing this data. Kona uses a log to track multiple dirty cache lines and sends it to the memory node in batches. Anil ▶anything else?◀

4.4.3 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 [28] for proper isolation in presence of multiple applications using such runtimes. TODO ▶also talk about performance isolation, does disaggregated design open up more avenues for security attacks?◀

5 DISCUSSION

Anil ▶WORKING HERE◀ Meta ▶Future challenges and opportunities in topics described above or any new ones that needs to be solved before disaggregation could become feasible/or to make it more adoptable. Look at [3] and assess what the paper got it right and what it did not.◀

1. Does memory disaggregation make sense for compute-intensive applications? What applications/use-cases is it the most suitable for? What are some apps where it would not work? Remote memory may not be that beneficial for compute-intensive applications as memory accesses get amortized and using disk may not be that bad. Then why did spark fail to run on Infiniswap? Because working set is big and it started thrashing?
2. What's the future of job scheduling like fastswap in the context of far memory?
3. How might the new hardware architectures e.g., emerging networking interconnects like OmniPath affect the design of such systems?
4. Future: how can remote memory be made to work with VMs/cloud tenants? Some works already consider hypervisors: [19].
5. Is remote memory-based app feasible performance-wise? Yes, since most of the memory tends to be cold and if caching is done right, it may not affect the performance that much.. look at the eval sections of various systems to prove this?
6. Could managed/userspace runtimes be the future of remote memory systems? Is far memory awareness really important in the longer term? What kind of adoption did NUMA-aware data structures see?
7. What's does the future look like for memory disaggregation? Is it gonna play out?
8. What server capabilities are reasonable? Should function shipping be allowed?
9. Which system comes close? Where can it get better? only infiniswap and legoos checks most boxes but it targets future hardware. i.e., what would you add to LegoOS? pure userspace runtimes are not it. but if co-design is a must, then may be kernel-based, ioctl runtimes are probably the way?
10. What interface works best for accelerators, keeping the hardware heterogeneity?
11. What is the bare minimum piece of runtime that must run locally? Perhaps parts of a monolithic OS like Linux can still go on remote memory leaving a very thin kernel required for supporting caching and

remote memory ops. What are the benefits? Is Yizhou's network disaggregation related?

12. Can we take lessons from somewhere regarding the transparency/generality vs performance debate?

13. Currently can't scale to more than rack-scale or a few racks due to interconnect latency. but as new technologies and runtimes within the server cut the latencies, it will perhaps allow more interconnect distance between the servers, and grow to an entire datacenter.

14. Hardware like Kona is good but handling the complexity of replication and security in the hardware might be hard?

15. Challenges that come with accelerators

16. Opportunities with programmable networking using it for memory and network load-balancing prevent congestion

5.1 Future

1. Semeru + Far Memory Data structures: currently semeru partitions and spreads the heap over multiple machines with no regard of data structure optimizations.

2. Kona/AIFM + Leap Prefetching may help with read I/O amplification as well.

3. Traditionally less focus on the intricacies of server-side memory management like avoiding fragmentation etc.

4. Security and Isolation, specifically performance isolation does not exist.

5. Need to extend it for virtualization solutions with proper SLO guarantees to serve the cloud model. Making it available for virtual machines through the hypervisor, there hasn't been any work in this direction. Through Hypervisor, would add an additional layer introducing overheads and gap between apps.

5. Wide ranging benchmarks? Spark failure shows that bad locality can hurt app performance beyond salvage due to thrashing. A standard benckmark with wide range of applications to properly compare across systems.

6 CONCLUSION

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

[16] 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

[17] 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

[18] 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

[19] 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

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

[21] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. *Latency-Tolerant Software Distributed Shared Memory*. 291–305 pages. https://www.usenix.org/conference/atc15/technical-session/presentation/nelson

[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, 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

[24] 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.

[25] 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

[26] Yizhou Shan, Shin Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. *SoCC 2017 - Proceedings of the 2017 Symposium on Cloud Computing* 15 (2017), 323–337. https://doi.org/10.1145/3127479.3128610

[27] 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

[28] 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

[29] 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

[30] 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