Remote memory in the age of fast networks

Marcos K. Aguilera
VMware

Nadav Amit
VMware

Irina Calciu
VMware

Xavier Deguillard
VMware

Jayneel Gandhi
VMware

Pratap Subrahmanyam
VMware

Lalith Suresh
VMware

Kiran Tati
VMware

Rajesh Venkatasubramanian
VMware

Michael Wei
VMware

ABSTRACT

As the latency of the network approaches that of memory, it becomes increasingly attractive for applications to use *remote memory*—random-access memory at another computer that is accessed using the virtual memory subsystem. This is an old idea whose time has come, in the age of fast networks. To work effectively, remote memory must address many technical challenges. In this paper, we enumerate these challenges, discuss their feasibility, explain how some of them are addressed by recent work, and indicate other promising ways to tackle them. Some challenges remain as open problems, while others deserve more study. In this paper, we hope to provide a broad research agenda around this topic, by proposing more problems than solutions.

1 INTRODUCTION

Disaggregated memory is an exciting technology proposed to improve memory utilization in cloud data centers [11, 19, 22, 25, 33, 39]. Its basic idea is to detach ("disaggregate") most of the memory of each machine, placing it on a common fabric, where it forms a *cluster memory* pool; this memory can be assigned to machines when and if they need it (Figure 1(a)). To realize disaggregated memory, new hardware architectures are under development in both academic and industrial settings (§5). While new hardware could be interesting, its cost could be prohibitive and its availability may be limited.

In this paper, we envision that disaggregated memory can be fully realized in *software alone*, without new hardware architectures, new standards, or new interconnects, using instead commodity off-the-shelf hardware available today. To clearly distinguish the software from the hardware approaches, we call the former *remote memory*, while reserving the term *disaggregated memory* for the hardware solution. With remote memory, each machine has a conventional

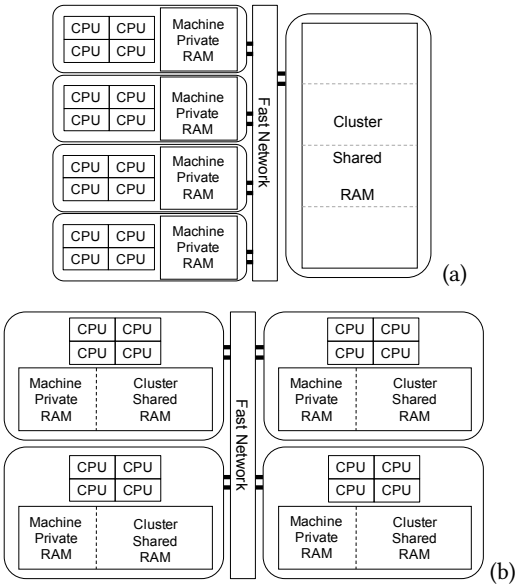

Figure 1: (a) Disaggregated memory. Memory is detached from machines and placed on a common fabric to all machines. Machines may retain some RAM, but the bulk of the memory is in the fabric. (b) Remote memory. Each machine has locally attached memory, with a part assigned to a pool of cluster memory. Machines are connected by a commodity network.

memory architecture and it contributes parts of its memory to the cluster memory pool. Machines that need memory then access the memory of another machine over the network (Figure 1(b)). Using the virtual memory subsystem, accesses to the remote memory are transparent, appearing like accesses to local memory. Remote memory brings many benefits, such as huge memories, better utilization, and more efficient data exchange (§6).

Remote memory is an old idea, similar to distributed shared memory (e.g., [10, 13, 31, 38, 42, 43]) and swapping to the network (e.g., [15, 20, 21, 30, 32]). These ideas were studied extensively 20 years ago, and now they are getting revived in different ways [22, 23, 48]. What is different today? In the 2000s, network latency was three orders of magnitude higher than memory latency (hundreds of

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3131612

Application	What is in remote memory
Graph processing	Large graph
Key-value cache	Key-value pairs
Key-value storage	Key-value pairs
Data analytics	Input and output of tasks
Database system	Tables, indexes
Statistical processing	Data sets
Machine learning	Input data, parameter sets

Figure 2: Applications of remote memory.

μs vs. hundreds of ns). Since then, network latencies have improved significantly, while memory latencies have not, so there is a gradual convergence of performance (§4). So, might remote memory work now?

It turns out that efficiently realizing remote memory requires overcoming several challenges, from virtual memory overheads, crashes of remote machines, sharing model, virtualization, scalability, and placement (§7). In this paper, we enumerate the challenges, discuss their feasibility, explain how some of them are addressed by recent work, and indicate other promising directions to explore. Currently, no system addresses all of the challenges. Some of the challenges remain open problems, while others have been studied but not extensively. We believe much more research is required in this topic, and we hope to provide a broad research agenda around it, by proposing more problems than solutions.

2 WHAT IS REMOTE MEMORY?

We target a data center environment within a public or private cloud. The data center is divided into smaller clusters of dozens of machines, such as a rack or a few racks, connected by a reliable, high-bandwidth, low-latency network. By low-latency we mean latencies close to $1\mu s$, as achieved by commodity RDMA networks today. Remote memory spans a cluster of such machines ("rack-scale") rather than the entire data center.

Remote memory provides the abstraction of a linear address space, which applications read and write at a byte granularity using the usual machine-level instructions for memory. We want remote memory to resemble local memory to applications as much as possible. However, remote memory should expose some of its differences to applications (e.g., how it handles failures, shares data, and places data §7), so we need new abstractions that are not transparent, but do not overburden programmers. Under the hood, remote memory is implemented by virtual memory paging, which we briefly recapitulate. When the application tries to access a page stored in another machine, it causes a page fault, which fetches the page over the network. The page is cached in local memory for the near future. If the page is modified, it is marked dirty and later written back to the remote machine.

To work well, this simple idea needs to overcome many challenges (e.g., what if the remote machine crashes with our memory?), which we cover in Section 7. We intend remote memory to serve a wide variety of applications that manipulate lots of data (Figure 2).

3 RELATED WORK

Remote memory is related to distributed shared memory (DSM), and there is much to be learned from this body of work. DSM is centered on transparent *sharing* (the "S" in DSM): it implements the abstraction of a shared memory system using a message-passing system, with the goal of running applications designed for the former system using the latter. This goal turned out to be elusive due to the difficulty of getting reasonable performance. 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). This sharing over time is distinctive from DSM's sharing for running multithreaded code; the sharing model for remote memory is an interesting research challenge (§7).

Recent work has started studying different aspects and use cases of remote memory. Infiniswap [23] proposes using remote memory as a fast cache for a local disk swap. Mojim [48] proposes using remote memory to safeguard the data stored in local non-volatile memory in case a machine breaks. These systems tackle some of the challenges of remote memory using solutions designed for their use cases, as we explain in Section 7. For example, Infiniswap tackles crashes of remote memory hosts by simply ignoring them; this is acceptable because Infiniswap stores only disposable data in remote memory (caches). Mojim asynchronously replicates data from one remote host to another, which again is acceptable for its use case. We are interested in a broader application of remote memory: to serve as a complete substitute for disaggregated memory.

Other work has studied the network requirements for disaggregated or remote memory [22]. That work mentions the software overheads of virtual memory without quantifying the problem.

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 fundamentally different from the abstraction of memory accessed via loads and stores, because they require developers to explicitly use them through special language-specific functions. For example, key-value storage systems require applications to invoke functions to get and put key-value pairs; database systems require functions to issue SQL queries; and RDMA requires functions to register memory and orchestrate requests and queues. In contrast, remote memory is accessed directly through *memory instructions*—the same loads and stores that are used for non-remote memory—providing a simple, transparent, convenient, and language-agnostic abstraction for developers.

For example, suppose a developer wants to read two locations a and b in remote hosts, and store their sum in a third remote location c. With remote memory, the developer can just write some simple code, such as $*c = *a + *b$ where a, b, and c are pointers to the desired locations; this is the same code whether the locations are local or remote. With a key-value interface, the developer must invoke $get(a)$ and $get(b)$, store the results in some variables, say x and y, and then invoke $put(c, x + y)$. With RDMA, the developer must initialize the context and register memory regions both at the application and at the remote hosts; establish connections to the remote hosts; exchange region keys; post RDMA-read operations on

			Network latency (ns)								
			Word, 100Gbps			Page, 100Gbps			Page, 25Gbps		
			now	↓20%	↓50%	now	↓20%	↓50%	now	↓20%	↓50%
			700	560	350	1020	880	670	1980	1840	1630
Memory latency (ns)	best	60ns	11.7	9.3	5.8	17.0	14.7	11.2	33.0	30.7	27.2
	typical	100ns	7.0	5.6	3.5	10.2	8.8	6.7	19.8	18.4	16.3
	nvm	200ns	3.5	2.8	1.8	5.1	4.4	3.4	9.9	9.2	8.2

Figure 3: Network latency (top), memory latency (left), and the ratio of network to memory (bottom right). Memory latencies are for the best commodity, typical commodity, and hypothetical non-volatile memory (nvm), speculated to be 2× slower. Network latencies are for transferring a word or a 4 KB page over a 100 or 25 Gbps network, for commodity networks now and speculative improvements of 20% and 50% on switching and NIC delays.

a and *b* and request the results to be placed in specially registered buffers; wait for the completion of the reads; store the sum in another specially registered buffer; post RDMA-write operations on *c* using the special buffer; and wait for the completion of the write.

While different, we believe the other abstractions (key-value, RDMA, etc) could be useful to implement remote memory. For example, an implementation of remote memory needs code to read and write pages remotely, and this code could use the get and put operations of a key-value storage system. Or one might opt for a direct RDMA implementation. Either way, the complexity of these abstractions are hidden by the implementation of remote memory.

4 WHY NOW?

Remote memory is now relevant because the latencies of memory and network are converging: eventually, there will be comparable costs for going to local memory or memory of a remote host. The latency of commodity memory has not been improving, as memory makers focus instead on higher densities and lower power [47]. Upcoming non-volatile memories, such as 3D XPoint, are expected to be slower than DRAM, perhaps by a factor of 2 or 3. Meanwhile, commodity network latencies have dropped to 700 ns roundtrip for a local area network [6]. New networking technologies, such as Omni-Path [5], will collocate NIC and CPU on the same package, and eventually on the same die, further reducing network latency. Figure 3 shows the ratio of network to memory latencies in various systems, some hypothetical. A ratio of 1.8 (word) or 3.4 (page) could become reality one day.

5 WHY A SOFTWARE SOLUTION?

To realize disaggregated memory, hardware solutions are currently under development both as research projects [33, 34] and industry efforts,such as CCIX [1], Gen-Z [3], OpenCAPI [7], and Omni-Path [5]. Given such hardware, why develop a solution that uses only commodity hardware and is implemented entirely in software? We believe this is warranted for many reasons. First, these hardware solutions are not yet available and may not materialize or gain traction. Second, even if they materialize, hardware solutions may not be ubiquitous or may have significant costs, especially in initial deployments. Third, from a scientific perspective, we want to understand if new hardware solutions are necessary and what are their advantages and drawbacks compared to a software solution in terms of performance, cost, energy consumption, etc. Fourth, we want to know if small hardware changes could bring big benefits to software solutions. To gain this understanding, we must develop the software solution.

Type	Benefit	Reason	Use case
Resource	Huge memory	Memory not limited by physical machine	New applications
Efficiency	Utilization	Memory not grabbed by machine	Low-cost memory provisioning
Efficiency	Cheap data exchange	Avoids marshalling and demarshalling to transmit data	Distributed data analytics
Resiliency	Decouples failure	Machine fails while retaining remote memory	HPC checkpointing, data management applications

Figure 4: Summary of benefits of remote memory.

Type	Challenge
Failures	Remote host crashes
	Network slow or congested
Virtualization	Virtual memory overheads
	Virtual machine indirection
Abstraction	Transparency level
	Proper sharing model - Lack of write ordering
	Non-uniform latency
Security	Remote host compromised
Resources	Local vs. cluster memory
	Remote memory allocation
	Memory placement
	Local memory management
Scalability	Control plane efficiency
	Memory metadata overhead

Figure 5: Challenges of remote memory.

6 BENEFITS

Figure 4 shows the benefits of remote memory, similar to those of disaggregated memory [11, 19, 22, 33]. First, machines have access to a huge memory, which enables new and existing applications to run at larger scale (e.g., in-memory databases and graph processing). Second, machines do not stash memory, reducing their provisioning cost (e.g., no need to provision each machine for the worst case). Third, an application can write data to remote memory; when it finishes, other applications can obtain its output, without the overheads of writing to a file system or serializing data; e.g., in map-reduce, the output of the mapper can be obtained directly from reducers over remote memory. Finally, application data survives machine crashes, because memory is separate from the machine; this has implications for HPC application checkpointing, database systems, and other systems that need durability.

7 CHALLENGES

To provide a complete alternative to disaggregated memory, remote memory must address a number of challenges that we now describe (Figure 5).

1. Remote host crashes. If a remote host fails, the application loses a large chunk of its remote memory, which could crash or freeze the application while the remote server is down.

Possible solutions. Broadly, this challenge admits two classes of solutions. First, expose failures to the application by allowing it to provide failure handlers—including handlers that just ignore the failure, as in Infiniswap. This is useful to handle disposable or reconstructible state (e.g., the output of deterministic computations). Second, mask the failure through redundancy, using replication or erasure coding. Doing this will require microsecond-speed mechanisms to maintain redundancy in the common failure-free case; for instance, erasure coding a 4K page needs to be competitive with copying the page. This could be achievable with FPGA support. Moreover, any solution needs to be memory efficient. This will require, for example, erasure codes with small space overheads.

Infiniswap [23] exposes rather than mask failures, but it can ignore them because remote memory is disposable: it is just a cache for the disk swap space, so that losing remote memory amounts to losing redundant cache entries. Mojim [48] replicates the non-volatile memory into remote memory synchronously, then replicates the remote memory at one host to another asynchronously. Because replication of remote memory is asynchronous, it does not add latency, but asynchronous replication can lose data.

2. Network slow or congested. Another challenge is performance degradation in the network due to congestion or pathological behavior [24]. This can lead to unpredictable application performance, which can cause unacceptable behavior for interactive applications (e.g., the system freezes momentarily), violations of service-level agreements, and timeouts. Even if the problem is rare, it might affect tail latencies, which are important for some applications.

Possible solutions. Some simple solutions are to prioritize network traffic and pre-allocate network bandwidth for remote memory. If there are many remote memory applications and not enough bandwidth for all of them, one could ensure network locality, by placing an application and its memory nearby in the network. Another possibility is to give application different regions of memory, each with its own characteristics: one region could have a higher network priority but smaller space, while another could have a larger space but lower network priority (hence higher jitter). Dynamic migration of remote memory might be necessary to balance the network load. And admission control might be required to prevent all applications from suffering. Alternatively, applications may need ways to cope with stalled memory accesses while congestion subsides.

3. Virtual memory overheads. Virtual memory paging was designed for disks that operate in large blocks and have latencies of 10 ms, but network transfers can have flexible sizes and are 10 000× faster. Thus, paging is inefficient and needs better mechanisms, as noted in [22, 34].

We show these overheads, the swap-in overheads, and a breakdown of device and software path overheads on a random-access workload with little memory and paging to local DRAM, implemented using pmem emulation [8] (an optimistic approximation of a fast network). The workload reads and writes locations at random from 4 GB of virtual memory. Figure 6(a) shows the throughput as we decrease the local memory one GB at a time. Performance drops by ≈ 44% each time, showing significant overheads. Figure 6(b) breaks down the overheads. Most of is determining what to swap out (41%), rather than writing the page (16%).

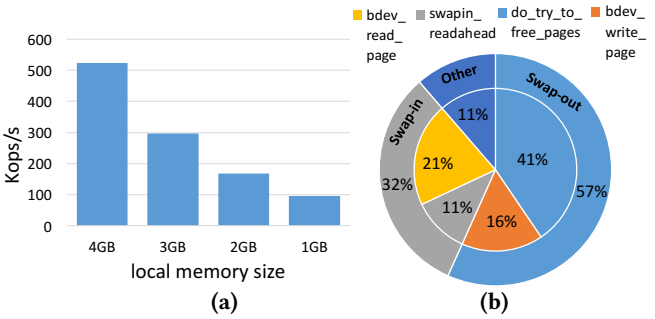

Figure 6: Paging overheads of virtual memory. (a) Paging to DRAM-based pmem block device when an application accesses 4 GB of data with a local memory of a different size. (b) Breakdown of overheads.

Possible solutions. There are techniques to overcome paging overheads (dedup [46], memory compression [44], latency hiding [37], etc.), but they are costly for remote memory, being designed for a *ms* scale. We need to overhaul the virtual memory subsystem for the *µs* scale, by removing data structure overheads, lock contention, and incorrect optimizations; by providing better prefetching-style optimizations; by cutting functionality (transparent huge pages, copy-on-write, etc) [18]; by matching the virtual memory transfer granularity (currently 4KB page) to the more fine-grained application accesses; and by leveraging new hardware features such as subpage-level protection [40]. We can benefit from new techniques to predict memory access patterns and thus avoid page faults, though doing so is notably hard [26, 41]. It might be possible to extend the hardware prefetcher to speculatively fault on pages. Alternatively, applications can provide hints about memory they intend to access soon. Those hints can be provided by the application developer, or they can be inferred by static code analysis coupled with dynamic instrumentation.

4. Virtual machine indirection. Virtual machines are prevalent in the environments we target: research shows that at least 70% of x86 server workloads are virtualized [9]. Thus, remote memory has to work with virtual machines. There are two approaches for supporting virtual machines. First, virtual machines can manage their remote memory independently of each other, as if they were separate hosts in the network. Second, virtual machines can coordinate the use of remote memory through the hypervisor. The first approach is the simplest, but it does not reflect the fact that virtual machines are actually sharing the resources of their host, including memory and network bandwidth. So, ideally, remote memory should be managed by the hypervisor, and the cluster memory can be managed by the virtualization control plane. The challenge is that this approach adds an extra level of indirection, which in turn imposes additional overheads for the virtual memory subsystem [46] and creates a semantic gap: the hypervisor does not see applications, only virtual machines, and so it is harder to use application-specific information to hide overheads [34].

Possible solutions. Besides mechanisms to reduce virtual memory overheads (discussed above), we need to find ways for the hypervisor to capture information about applications, either implicitly by inferring their behavior through the virtual machine, or explicitly

by relying on hints provided by the application or by the virtual machine operating system.

5. Transparency level. In theory, remote memory can be fully transparent to applications, looking exactly like local memory. In practice, several of the remote memory challenges might require help from the application. In fact, application awareness could address all other challenges of remote memory, but awareness amounts to intrusion, and the challenge is to find the right balance.
Possible solutions. One solution is to specialize remote memory for a use case. For example, Infiniswap uses remote memory solely as a cache that need not be accessible and can be evicted at any time. Mojim uses remote memory to safeguard the local contents of non-volatile memory. These use cases allow Infiniswap and Mojim to provide a fully transparent abstraction.

When full transparency is not possible, it is useful to distinguish different ways in which applications can provide hints: (1) memory allocation (2) pointer type annotation, and (3) memory accesses. With (1), memory allocations can be annotated with the type of memory that the application expects. For example, malloc can be extended with a special flag to ask for memory that is replicated, or that does not suffer from jitter, or both. These types of annotation may not be intrusive when there are relatively few places in the code where memory is allocated, but determining which allocations are remote is a manual task that requires application-specific knowledge. With (2), each pointer type indicates what the application expects from the memory pointed to by the pointer (memory reliability, speed, jitter, etc). A type checker can ensure that a pointer is assigned only to memory that is consistent with the requirements of the type. For example, a pointer might be allowed to point only to memory that is fault tolerant. Pointer type annotations by themselves are not intrusive, but intrusiveness might arise from the type system, which prohibits assigning incompatible pointer types. With (3), the application or the compiler needs to annotate a memory access, so that the remote memory system can take the required actions. For example, when an application changes its working set, it might annotate the next access to avoid a page fault: the annotation can proactively cause the memory system to fetch the required page before it gets accessed. Memory access annotations can be very intrusive if they need to be done often and by the developer, so it would be ideal either to do them automatically using the compiler, or to reduce the cases when they are needed.

6. Sharing model. Cache coherence is expensive even within a machine, and for remote memory its cost is prohibitive. Thus, we need to find a reasonable *memory sharing model* to expose to applications, without coherence. This model indicates when applications can share data, at what granularity sharing occurs, and what behaviors application observe if sharing occurs (e.g., what happens if applications concurrently read and write the same location). Defining an intuitive sharing model is generally hard.
Possible solutions. An approach is to limit remote memory to private data, where sharing is prohibited. This approach avoids the problem, and it is well suited for certain use cases of remote memory (e.g., Infiniswap and Mojim).

A more general solution is to allow sharing but not simultaneously. At any time, memory is accessed exclusively by a single host, but across time it can be accessed by many. This approach requires some way for applications to signal when they want to start and stop accessing the remote memory. This approach is useful for applications in which one host produces some output in remote memory, which is later consumed by another host (e.g., map-reduce, parameter server in machine learning, graph processing framework, etc).

7. Lack of write ordering across hosts. Remote memory could be spread over many remote hosts, each with a different roundtrip latency, as hosts may be slow or connected to different racks. Thus, write-backs to remote memory can be reordered. Reordering becomes a problem when the data is subsequently read (by the same or different hosts) and the correct ordering is not respected. For example, an application might set some data value and a pointer, but later see the pointer to invalid data.
Possible solutions. DSM avoided this problem by enforcing ordering with appropriate protocols, but this can be costly. Another approach is to restrict an application to use just one remote host for memory, as in Mojim, so that it is impossible to reorder writes across remote hosts. (Within a remote host, writes can be ordered easily.) A third approach is to allow reordering, for applications whose semantics support it, e.g. Infiniswap. However, none of these solutions are generally applicable.

8. Non-uniform latency. As hosts have different roundtrip latencies, some parts of memory are faster than others. This can be disruptive, as an application can be bottlenecked by the slowest memory that it has, thereby wasting the higher performance provided by parts of its memory.
Possible solutions. This problem is similar to NUMA architectures, which have multiple sockets and a given memory location can be accessed more quickly from one of the sockets. But remote memory adds more tiers (e.g., local, same rack, different racks). One approach is to use the existing operating system mechanisms for NUMA [29], extending them for more tiers. Another approach is to expose the memory speed to applications, which can use appropriate data structures and layouts to optimize their performance. Alternatively, the system may need to carefully choose where to place memory (see "memory placement" below), to avoid using remote hosts far from the application.

9. Remote host compromised. Applications assume that memory is private and safe; they store confidential information in memory: plaintext passwords, keys, cookies, capabilities, etc. This assumption is met by the hardware and kernel designs: the interconnect (memory bus) is physically secured, and the kernel protects pages and clears them when they are reclaimed. Remote memory increases the attack surface, being vulnerable to attacks on the local machine, the remote machines, and the network. Even if an attacker does not control these components, it can launch side-channel and denial-of-service attacks.
Possible solutions. We envision two approaches. First, we could assume that memory is no longer safe and private, so we encrypt the data in remote memory; this can be done directly by applications only for its sensitive data, or transparently by the virtual memory subsystem for all data; the latter will require microsecond-scale encryption mechanisms. Second, we can maintain the assumptions by strengthening the security of the larger system, to compensate for the larger attack surface. In particular, it is important to

isolate consumption of resources—such as network, memory, and processor—across applications in every machine. This isolation needs to be strong, to avoid side-channel attacks, say by static partitioning resources; work-conserving sharing, which yields unused resources of one application to another, can leak information.

10. Local vs. cluster memory. Each host must decide on a split between local and cluster memory (Figure 1(a)), depending on how much local memory it needs for its own applications, in addition to space for kernel data, file caches, stacks, etc. Local memory is stashed away for local consumption, so it should not be too large, but if it is not large enough, the system may run out of memory and become unstable. In particular, the system needs to pre-reserve enough local memory to be able to handle its own local metadata and to communicate with remote machines to request more memory.

Possible solutions. A simple solution is to adopt a static allocation that reserves a reasonable amount of local memory (e.g., 2 GB) and leaves the rest for cluster memory. Dynamic allocation is more challenging: if a host gives up local memory to the cluster, the host may not be able to reclaim the memory when it needs it, because the memory may be in use. One way to address this problem is to allow the local host to deallocate cluster memory whenever it wants, even if the memory belongs to another application. This approach is taken in Infiniswap, which is acceptable because cluster memory is used only for caches. But this does not work for all applications. A general solution requires mechanisms to migrate the data in cluster memory to another host, so that it can be reclaimed locally.

11. Remote memory allocation. Hosts may compete for memory, and we need mechanisms to track allocations, provide reservations, enforce quotas, ensure memory is available to critical applications when they need it, and broadly enforce cluster-wide allocation policies (e.g., a batch application can use more than 64 GB of memory only if no interactive applications are running).

Possible solutions. The simplest approach is to centralize the problem, by requiring allocations across the cluster to go through a host that manages memory. To improve performance, allocations can be done in large slabs (e.g., 64 MB), so that they are infrequent, while application libraries can divide each slab locally and provide smaller and finer-grained allocations through library functions (e.g., malloc). To provide fault tolerance and availability, the host managing memory can be replicated.

12. Memory placement. When the system allocates remote memory, it must choose which remote host(s) will actually store the memory. This choice might depend on the network latency (nodes within a rack have lower latencies), bandwidth (some paths may be less congested), and memory availability at each host (it may be desirable to balance memory usage across hosts). The challenge here is that of mechanism and policy.

Possible solutions. The simplest mechanism is to centralize the decision of placement, much like we can centralize the decision of allocation (see above). Deciding on a policy is much harder, because there can be many different opposing requirements. One solution is to model placement as an online optimization problem, where some of the requirements are taken as constraints (e.g., memory placed on a host cannot exceed its capacity), while others are part of an optimization function (e.g., nearby placements are given a higher utility).

Another approach, used in Infiniswap, is to use the power-of-two choice to balance the allocation across the cluster using a decentralized scheme. This approach, while simple, provides little flexibility in the choice of placement policies.

13. Local memory management. Each host manages its physical memory to reduce fragmentation, share zero pages, migrate pages for NUMA affinity, etc. These tasks require moving and remapping physical pages. When a local page belongs to a remote host, who should do these tasks and how?

Possible solutions. We believe these tasks are best handled by the machine hosting the physical memory, rather than the remote machine. But there is a trade-off here: doing it locally requires an extra level of virtual memory local to each machine, which adds overheads on modern RDMA-based NICs (e.g., IOMMU TLB misses [35]).

14. Control plane efficiency. As we expand the cluster, scalability becomes a concern. There are scalability issues of the network fabric: it may be infeasible to maintain pairwise connections between all machine pairs [16, 28]. There are also scalability issues of the control plane of the remote memory, which monitors failures, keeps the cluster membership (what hosts provide remote memory), and tracks remote memory allocations.

Possible solutions. We may be able to leverage off-the-shelf solutions for control planes [2, 27]; an experimental study is required to validate their performance for this use. If performance is insufficient, one might need to optimize and specialize them for remote memory, or make them work over RDMA [12].

15. Memory metadata overhead. Very large memories create significant page metadata (e.g., lock, refcount, flags, lru list pointer, cgroup pointer, page cache mapping), which needs to be stored locally even if memory is remote. This metadata may not fit in local memory. We need ways to combine or compress metadata, or memory mechanisms that can omit metadata.

Possible solutions. One approach, taken in Infiniswap, is to manage remote memory in slabs that are much larger than the page size, so that the system need only keep one set of metadata for each slab. Another approach is to realize that large non-volatile memories face similar challenges and the community is actively engaged in solving this problem. Remote memory may be able to leverage those solutions.

8 CONCLUSION

Remote memory is a promising idea whose time has come in the age of fast networks. This paper enumerates its challenges and indicates directions to explore, setting a broad research agenda around this topic.

9 ACKNOWLEDGMENTS

We thank Mihai Budiu, our shepherd Patrick Stuedi, and the anonymous reviewers for many useful comments.

REFERENCES

[1] CCIX: cache coherent interconnect for accelerators. http://www.ccixconsortium.com. Accessed: 2017-05-05.

[2] etcd 3.1.7 documentation. https://coreos.com/etcd/docs/latest. Accessed: 2017-05-05.

[3] Gen-Z draft core specification, December 2016. http://genzconsortium.org/draft-core-specification-december-2016.

[4] InfiniBand. http://www.infinibandta.org/content/pages.php?pg=about_us_infiniband. Accessed on 2017-01-24.

[5] Intel Omni-Path. http://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html. Accessed on 2017-01-24.

[6] Mellanox Connect X4. http://www.mellanox.com/page/products_dyn?product_family=201&. Accessed on 2017-01-24.

[7] OpenCAPI consortium. http://opencapi.org. Accessed: 2017-05-05.

[8] pmem.io persistent memory emulation in DRAM. http://pmem.io/2016/02/22/pm-emulation.html.

[9] Magic quadrant for x86 server virtualization infrastructure. https://www.gartner.com/doc/2788024/magic-quadrant-x-server-virtualization, 2014.

[10] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

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

[12] J. Behrens, S. Jha, M. Milano, E. Tremel, K. Birman, and R. van Renesse. The Derecho project. https://derecho-project.github.io.

[13] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 168–176, Mar. 1990.

[14] N. Carriero and D. Gelernter. The S/Net's Linda kernel (extended abstract). In *ACM Symposium on Operating Systems Principles (SOSP)*, page 160, Dec. 1985.

[15] D. Comer and J. Griffioen. A new design for distributed systems: The remote memory model. In *Usenix Summer 1990 Technical Conference*, pages 127–136, June 1990.

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

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

[18] J. Edge. DAX, mmap(), and a "go faster" flag. https://lwn.net/Articles/684828/. Accessed on 2017-01-24.

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

[20] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 201–212, Dec. 1995.

[21] M. D. Flouris and E. P. Markatos. The network RamDisk: Using remote memory on heterogeneous nows. *Cluster Computing*, 2(4):281–293, Oct. 1999.

[22] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 249–264, Oct. 2016.

[23] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 649–667, Mar. 2017.

[24] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over commodity ethernet at scale. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 202–215, Aug. 2016.

[25] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *Workshop on Hot Topics in Networks (HotNets)*, pages 10:1–10:7, Nov. 2013.

[26] M. R. Hines, A. Gordon, M. Silva, D. Da Silva, K. Ryu, and M. Ben-Yehuda. Applications know best: Performance-driven memory overcommit with Ginkgo. In *Cloud Computing Technology and Science (CloudCom)*, pages 130–137, Nov. 2011.

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, June 2010.

[28] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 295–306, Aug. 2014.

[29] A. Khandual. [RFC 0/8] Define coherent device memory node. https://lkml.org/lkml/2016/10/24/19. Accessed on 2017-01-24.

[30] S. Koussih, A. Acharya, and S. Setia. Dodo: A user-level system for exploiting idle memory in workstation clusters. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 301–308, July 1998.

[31] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, Nov. 1989.

[32] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over InfiniBand: An approach using a high performance network block device. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10, Sept. 2005.

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

[34] K. T. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 189–200, Feb. 2012.

[35] M. Malka, N. Amit, M. Ben-Yehuda, and D. Tsafrir. riommu: Efficient iommu for i/o devices that employ ring buffers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 355–368, Mar. 2015.

[36] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference (ATC)*, pages 103–114, June 2013.

[37] G. Natapov. Asynchronous page faults - AIX did it. http://www.linux-kvm.org/wiki/images/a/ac/2010-forum-Async-page-faults.pdf. Accessed on 2017-01-24.

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

[39] P. S. Rao and G. Porter. Is memory disaggregation feasible? A case study with Spark SQL. In *Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 75–80, Mar. 2016.

[40] R. Sahita, V. Shanbhogue, G. Neiger, J. Edwards, I. Ouziel, B. Huntley, S. Shwartsman, D. Durham, A. Anderson, and M. LeMay. Method and apparatus for fine grain memory protection, Dec. 2015. US Patent App. 14/320,334.

[41] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone. Application level ballooning for efficient server consolidation. In *European Conference on Computer Systems (EuroSys)*, pages 337–350, Apr. 2013.

[42] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 174–185, Oct. 1996.

[43] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 297–306, Oct. 1994.

[44] I. C. Tuduce and T. R. Gross. Adaptive main memory compression. In *USENIX Annual Technical Conference (ATC)*, pages 237–250, Apr. 2005.

[45] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems, Nov. 1994.

[46] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 181–194, Dec. 2002.

[47] S. Woo. DRAM and memory system trends. In *Keynote International Symposium on Memory Management (ISMM)*, Oct. 2004.

[48] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–18, Mar. 2015.