FluidMem: Open Source Full Memory Disaggregation

by

Blake Caldwell

B.S., University of Colorado at Boulder, 2014

M.S., University of Colorado at Boulder, 2015

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2019

This thesis entitled:
FluidMem: Open Source Full Memory Disaggregation
written by Blake Caldwell
has been approved for the Department of Computer Science

Prof. Richard Han

Prof. Eric Keller

Prof. Sangtae Ha

Prof. Kenneth Anderson

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Caldwell, Blake (Ph.D., Computer Science)

FluidMem: Open Source Full Memory Disaggregation

Thesis directed by Prof. Richard Han

To satisfy the performance demands of memory-intensive applications facing DRAM shortages, the focus of previous work has been on incorporating remote memory to expand capacity. However, the emergence of resource balancing as a priority for cloud computing requires the capability to dynamically size virtual machine memory up and down. Furthermore, hardware-based or kernel space implementations hamper flexibility with respect to making customizations or integrating the continuing open source advancements in software infrastructure for the data center. This thesis presents an architecture to meet the **performance**, **bi-directional sizing**, and **flexibility** challenges of memory disaggregation in the cloud. The implementation, called FluidMem, is open source software [35] that integrates with the Linux kernel, KVM hypervisor, and multiple key-values stores. With FluidMem, a virtual machine's local memory can be transparently extended or entirely transferred to a remote key-value store. By fully implementing the dynamic aspect of data center memory disaggregation, FluidMem allows a VM's footprint to be precisely sized, expandable for application demands, but it leaves cloud operators with a non-intrusive recourse if memory becomes scarce. **Full memory disaggregation** in FluidMem enables the local memory footprint of a Linux VM to be scaled down to 180 pages (720 KB), yet still accept SSH logins. A user space page fault handler in FluidMem uses the userfaultfd mechanism in the Linux kernel to relocate any page of the VM to remote memory and outperforms swap-based paging. Page fault latencies via FluidMem to RAMCloud [84] are 40% faster than the RDMA remote memory swap device in the Linux kernel, and 77% faster than SSD swap. FluidMem's remote memory expansion performance with three key-value backends is evaluated against swap-based alternatives for a MongoDB [69] workload and the Graph500 [73] benchmark.

Acknowledgements

The unending support of my family and friends was critical to my completion of this thesis, from my mom's encouragement to challenge myself and begin the Ph.D. program, to the makeshift office on a ping-pong table in my dad's basement that allowed me the space to put my thoughts down on paper. My advisor Prof. Richard Han's encouragement, early on, to make FluidMem into something bigger with real impact was formative. I have gratitude for the other members of my committee, especially Prof. Sangtae Ha and Prof. Eric Keller, who lent their expertise to the thoughtful evaluation of my work, developing the rigor necessary for this thesis.

I was fortunate to have the opportunity to work with Youngbin Im on optimizations of FluidMem. His help with asynchronous evictions and a prefetch mechanism contributed in large part to FluidMem's performance. I would also like to acknowledge the students who were involved with FluidMem at various points during its development. Daniel Zurawski's help with the operation of the testing environment was well-timed and exceptional. Moreover, the contributions of William Mortl and Kannan Subramanian helped propel FluidMem development early on.

Finally, work on FluidMem was supported by the National Science Foundation under Grant No. 1337399.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

As fast random access memory continues to increase in speed and capacity, there still exist applications that demand more of it then can be supplied by a single server. Whether for scientific [61, 119], business [29], data storage [69], or data analytic purposes [118] these memory-intensive scale-up applications perform better if a larger percentage of their dataset can be stored in memory. Due to the infeasibility of storing the entire working set in memory, such applications make use of lower tiers of the memory hierarchy like disk storage, or if it's available, remote memory. A second-order concern after performance is for an application to successfully complete, even if it failed to estimate how much memory it will use and exceeds DRAM capacity. Memory-intensive applications rely on the virtual memory abstraction of the operating system to transparently page memory in and out of alternative storage to protect against out-of-memory (OOM) errors. Furthermore, in today's cloud computing environments user applications are run in virtual machines for portability, isolation and resource management reasons. When considering a cloud data center with racks of servers, cloud operators have a desire to minimize inefficiencies in how memory is used by VMs and be able to rebalance workloads without service interruption. Similarly, cloud users that have subscribed to the model of elastic resources need the ability to dynamically size up and down VM memory. If operating systems can meet these challenges and be integrated into existing cloud environments, then a model of memory consumption analogous to elastic disk storage can finally be realized.

Since the amount of memory on a single system is limited by factors such as heat dissipation,

timing variances and cost, fast memory capacity has never been able to keep up with application needs. The prospect of transparently extending memory capacity would provide wide-reaching performance benefits without changes in the programming model. This has resulted in an accumulating body of research surrounding remote memory.

Some of the first attempts to use remote memory were hardware platforms that combined processing power from multiple servers with a coherent address space composed of their collective memory capacity. However, large shared memory systems in hardware were costly, and were only installed at a small number of sites. As an alternative, software systems known as a Distributed Shared Memory (DSM) systems were actively researched from the mid-1980s onwards [7, 9, 30, 31, 33, 36, 54, 66, 89, 103]. These systems attempted to arbitrate access to the same memory storage from multiple workstations and solve the problem of ordering constraints and global cache coherency through networking validation protocols. Since the update of a particular data value could require notifications to broadcast before the program could continue, the latency of a single write could be as long as several network round trip times. Thus, network speed was the limiting factor for the performance of global shared memory systems.

Two relatively recent hardware trends have significantly increased the usefulness remote shared memory. First, increasing core counts on processor dies enabled cache-coherency to be handled in hardware for larger bundles of compute power. Cache-coherent NUMA meant that upwards of 20 compute units could share a single coherent address space, similar to what the DSM systems of the past tried to create. For applications with computational demands that could be satisfied on a single server, the problem of remote memory access no longer involved invalidation messages and lowers the number of network roundtrips for remote memory access to one.

Second, network speeds have continued to improve as new data center networking technologies such as DPDK [26] and RDMA have dropped latencies to 5-10μs for 4 KB messages [45]. While accessing memory over the network is still slower than DRAM accesses, which take about 25 ns for 4K reads on a NUMA system, other research has already leveraged RDMA to make remote memory an effective substitute for hard disk storage [56]. In data center and cloud environments with varied

workloads by user and over time, the underutilized resources of one server are a tantalizing target for memory extension of another server in need. Research systems of the past approach this problem by co-opting unused memory as cluster memory that can be shared among other servers [18, 95]. This provides a path to an increase in server memory utilization, but performance concerns make for complex software designs. Also, the opportunistic nature of cluster memory makes it difficult to guarantee performance through transient spikes in resource (memory or CPU) demand.

An evolution of ideas from cluster memory has emerged recently, called *memory disaggregation* [22, 28, 37, 38, 39, 58, 59, 62] which differs from the above-mentioned research in that it seeks to decouple memory used by applications from physical server DRAM, thus removing limitations on sizing and placement. In the disaggregation model, computational units can be composed of discrete quantities of memory from many different servers. It promises better resource allocation efficiency, but also allows the possibility of creating VMs larger than would be possible on any one server. However, works in memory disaggregation have not explored the question of how allocated VM memory should be scaled down. The reason for scaling down could be energy efficiency or cost efficiency, similar to purposes of live-migration to smaller VMs [105, 17]. A fine-grained downsizing mechanism would be a lightweight solution in comparison and allow for automating the discovery of VM working set sizes [77].

This thesis refers to *full memory disaggregation* as a true implementation of the theory that VM memory can be separated from host memory. The implementation of full memory disaggregation should allow cloud operators to perform uncooperative memory resizing for purposes of resource rebalancing and implementing memory service tiers. We note that the balloon drivers in KVM, Xen, and VMware can reduce the footprint, but are slow [112], and require the cooperation of the guest [111]. Even so, the balloon has a maximum size since the operating system needs some memory to keep the VM running.

The flexibilities of full memory disaggregation are particularly relevant to the virtualized environments present in cloud data centers today. In these environments, the virtualization layer provides a convenient abstraction for inserting remote memory and decoupling the VM from physical

memory. Many of the concepts discussed in this thesis will be applicable to memory disaggregation on bare-metal servers, but since memory address translation in Linux takes place at least partly in hardware, without a virtualization layer, the design decisions around intercepting page faults might differ. However in today's data centers, it is the user applications running in VMs that need the flexibility and performance benefits of memory disaggregation.

Finally, the applications that are at the center of this thesis's goals are known as scale-up applications. They are applications that increase memory working set size proportionally to dataset size. Thus, as dataset sizes natural grow, scale-up applications are bound to eventually require more memory than is available on a single node. These applications may be ill-suited for scaling-out in a distributed fashion either because the computation is serial in nature, or the code base is viewed as legacy, where the effort or expertise required for parallelization is infeasible. A user running a scale-up application on the cloud may be most concerned about increasing memory capacity, but when the workload requirements decrease from their peak, cloud operators could also benefit from being able to reduce the VM footprint. The former benefit has been explored in DSM and previous memory disaggregation works. However, full memory disaggregation is still an unmet need.

This thesis investigates the path towards full memory disaggregation given the context of virtualized cloud data centers with users running scale-up applications. Realizing memory disaggregation with all open source components is a priority so that pieces of the system can be customized for heterogeneous deployments. Recently there has been notable innovation within the open source community with developments in low latency key-value stores [84], and a new mechanism for page fault handling in user space [109]. This thesis will make the following contributions:

(1) **Performance**

   (a) A user space page fault handler implementation that outperforms swap-based paging. Page fault latencies via FluidMem to RAMCloud are 40% faster than an RDMA NVMe over Fabrics [79] remote memory swap device and 77% faster than SSD swap.

   (b) Demonstration of FluidMem outperforming swap-based remote memory used by ex-

isting memory disaggregation implementations in a MongoDB [69] workload and the Graph500 [73] benchmark

(2) **Completeness of memory disaggregation**

(a) A memory architecture that allows any page of the VM to be relocated to remote memory. With the memory footprint reduced to 180 pages (720 KB) a VM can still respond and open up an SSH shell

(b) The extension of memory disaggregation to any operating system virtualized by KVM [50] or emulated by Qemu [87]

(c) Demonstration of FluidMem full memory disaggregation completing the Graph500 benchmark with only 21% of the working set resident local memory

(3) **Flexibility**

(a) An open source implementation that integrates with the Linux kernel, KVM hypervisor, and key-values stores RAMCloud [84] and memcached [65]

(b) A template interface to allow for additional key-value backends to be supported through open source contributions

(c) Configurable optimizations such as asynchronous reads and writes, buffer pre-allocation, and a prefetch mechanism

(d) Integration with memory hotplug to allow the addition of remote memory in emergencies to avoid OOM errors.

The remaining chapters of this thesis are organized as follows. Chapter 2 begins with a review of related remote memory research. Next a prototype implementation using RAMCloud for disaggregation in Stateless Network Functions is explored in Chapter 3. In Chapter 4 the architecture for providing full memory disaggregation will be presented along with its implementation in FluidMem. Chapter 5 follows with an analysis of the micro-benchmark performance of user space page fault handling and evaluation of standard cloud applications running on FluidMem. Chapter 6 is a discussion of FluidMem's integration with existing data center models, opportunities for

improvement, and emerging trends in the cloud data center. This thesis concludes with a summary Of FluidMem's contributions and priorities for future work in Chapter 7.

Chapter 2

Related Work

	Performance	By-directional sizing	Flexibility	Transparency	Note
DSM	X	X	*	*	User space implementations were flexible. Kernel space implementations were transparent.
PGAS	✓	X	✓	X	Require app. code modification
Key-value stores	✓	X	✓	X	Require app. code modification
Single System Images	✓	X	X	✓	Major operating system rewrite
Remote paging	✓	X	X	✓	Kernel changes or limitations of swap space
Memory disaggregation (existing)	✓	X	X	✓	Only partial memory disaggregation
FluidMem	✓	✓	✓	✓	

Table 2.1: Remote memory approaches and their limitations in terms of providing full memory disaggregation in the cloud (X = violation, checkmark = compliance, * = see note).

Leveraging remote memory over a network for the purpose of augmenting local memory has been investigated in many prior works. The goals of these have varied widely, and have changed over time with respect to the operating system and network technology used. This chapter will review the different memory models used to integrate remote memory and other research areas that are relevant to full memory disaggregation. An overview of each memory model and their scoring relative to criteria necessary for full memory disaggregation is shown in Table 2.1

2.1 Memory models

When considering methods for expanding the available memory space for a general application on a multiprocessor system, those that preserve a model of sequential consistency are the most desirable. This is a very restrictive model where the system must maintain: 1) program order of

operations on a single processor 2) a single sequential order between all processors [1]. Memory systems that do not preserve this model are unsuitable for implementing full memory disaggregation because they would require modifications to guest VMs or applications. Relaxing this memory model in the name of better performance will increase code complexity in the applications that run on them. Use cases for cloud computing are concerned with a broad scope of applications, those that are coded in a general way, and not requiring knowledge of specialized infrastructures. Thus, application changes to incorporate remote memory, or even the transparent interposition of standard C libraries [66] are not acceptable in an implementation suited for the cloud.

2.1.1 DSM

Early supercomputers such as the Cray-1 [101] created large shared memory address spaces through hardware memory buses. These systems were extremely limited in physical size as evident by the circular design of early Cray supercomputers. To expand beyond the boundary of a single node and use memory from adjacent nodes would mean using a network instead of a hardware memory bus. Systems were able to successfully do this, but to mitigate network latencies, they utilized specialized networking hardware. These early implementations of multi-node shared memory were known as message passing or Distributed Shared Memory (DSM) systems [2, 21, 49, 51, 52, 98]

Among the earliest implementations of a single address space via network message passing was the MIT ALEWIFE system [2]. It's interesting to note that some of its optimizations are still relevant to modern systems and used in FluidMem, such as multi-threading to hide latency. However, these optimizations were of even greater importance at the time because of the relatively high network latencies. DSM hardware systems are the most extreme solution to the remote memory problem, providing the highest memory capacities and best performance, but also costing the most. The SGI UV 3000 [99] can provide up to 64 TB of cache-coherent memory. Such a system would not be found in a cloud environment because of the high cost.

2.1.1.1 Software DSM

Software DSM systems such as Ivy [54], Munin [7], Mach [36], Mirage [31], Clouds [89], and others [9, 30, 33, 66, 103] provide a single address space, like hardware DSM systems, but added protocols for sharing and invalidation into the operating system. Network latency and reliability posed a problem for the design and performance of these protocols. For example in [30] disk I/O was required for certain pages before sending the request over the network.

Typically, software DSM systems modified the lower levels of the operating system (the kernel) to introduce remote memory to user applications through system calls. Unfortunately, many of these implementations were on operating systems that did not survive to the present day (LOCUS [32], PS/2 [33], OSF/1 [30]).

Implementations in user space also existed, but they required applications to explicitly call out when data should be stored in remote memory. Some of the first DSM systems Ivy [54] and TreadMarks [46] were implemented in user space and had their own APIs.

2.1.2 PGAS

An optimization to DSM that avoids the high overhead from global coherency traffic is to split a global coherency domain into partitions. These systems, known as Partitioned Global Address Space (PGAS) systems, allow for the existence of multiple, separate key domains [13, 15, 75]. In effect, this limits the number of nodes that must maintain coherent copies of data. PGAS implementations give applications more explicit control over data placement, but that is also their limitation in the context of this thesis. Since these approaches to remote memory require code modifications to leverage remote memory, their applicability to cloud workloads is severely limited.

2.1.3 Key-value stores

Distributed in-memory key-value stores can be viewed as a type of PGAS that requires ordering, but not coherency. Unless replication is used, there may only be one copy of each data value, so invalidation protocols may not even be needed. Some in-memory key-value stores are designed

for commodity networks [65, 91], while others are designed for high-speed network transports using Infiniband, DPDK, or RDMA over Ethernet [27, 84, 67]. The use of network transports and a physical medium that can make assumptions about lossless operation is critical to lowering the latencies of remote memory.

2.1.4 Single system images (SSI)

Single system image implementations provide the abstraction of a machine composed of aggregated resources of multiple nodes [71, 72, 81]. These are heavyweight modifications of the operating system. The patches are often not maintained because of the significant work keeping up with the development of the Linux kernel. They implement DSM to provide a single address space with coherency guarantees, just like a system would expect if it was running on the same hardware. Linux SSI implementations [47, 72, 81] aren't true hypervisors in the sense that they run individual processes. As a result, they don't provide the same level of security and isolation as a VM. So they are not suitable to a cloud environment built around VMs.

2.1.5 Remote paging

The following systems offer an application-transparent interface of accessing remote memory by interfacing with the operating system's swapping mechanism or virtual memory management of memory pages.

2.1.5.1 Remote-backed swap space

Most operating systems have a swap mechanism for moving pages from main memory to disk and vice versa. Since the existence of this device is almost universal, and there is already a convenient translation layer between memory pages and block sectors, it makes an attractive target for remote memory. The transparent use of remote memory by unmodified applications can be realized using a custom memory-backed block device configured as the swap device [12, 34, 56, 63, 76, 41, 56, 38]. Observing that the performance of data center networks lies in between that of

DRAM and disk, several works have shown that emulating disk storage with a memory store on a remote machine offers performance benefits. Some works specifically targeted a virtualization use case where the VM is configured with a remote-memory backed swap device [41, 76]. Early designs paged over ATM [63] and Ethernet [34, 76], while more recent works have made use of kernel-bypass RDMA transport over an Infiniband network for improved performance [38, 56]. While replacing the swap device does succeed in inserting a layer of remote memory in the memory hierarchy between DRAM and disk, it is not the same as increasing the amount of addressable physical memory. This distinction is discussed in 4.1.1.

2.1.5.2 Non-swap paging

Other research recognized that the hypervisor could be modified to transparently present remote memory to a guest VM. By modifying the hypervisor's memory management, guest virtual addresses can be translated to remote memory addresses instead of a host physical addresses. Approaches that provide a Single System Image (SSI) for a guest VM by modifying the hypervisor [14, 47, 62, 72, 81, 96, 107] extend visibility of remote memory to the VM. Like the other hypervisor approaches vNUMA [14] provides a coherent distributed address space, but was only designed for the Itanium architecture, and the paper does not describe how the custom hypervisor manages running multiple VMs.

Commercial systems ScaleMP [96] and TidalScale [107] require installation of a proprietary OS and hypervisor. Additionally ScaleMP will only work with an Infiniband network, which may not be practical in all cloud environments. FluidMem achieves its best performance over an Infiniband network, but also supports the TCP over Ethernet transport through the memcached backend.

2.1.6 Memory disaggregation

Rack-scale memory disaggregation approaches [22, 28, 37, 39, 58, 59] are primarily prototypes that assume the presence of specialized hardware to store and retrieve pages distributed between

servers in close proximity. There is no support in these disaggregated architectures for the pre-existing heterogeneous hardware of today's data centers. Of note is that the assumptions of [37] match those of this thesis, including the cache coherency domain being limited to a single compute blade, page-level remote memory access, and logical resource aggregation by VMs. They evaluated the performance of several applications including Spark [117], COST [64], and memcached [65] in a disaggregated data center environment. Their conclusion on the network requirements for data center disaggregation was that network latencies of 3-5us between a VM and disaggregated memory stores were necessary for minimal performance degradation. Rao et al. investigated whether the specific application workload of Spark SQL would be suitable for full memory disaggregation in [90]. Their conclusion was that memory bandwidth would be sufficient, but they did not include an analysis of latency requirements.

A memory disaggregation approach that is implemented in software is Infiniswap [38]. Even though the evaluation of Infiniswap was done on bare-metal hardware with containers, the concept could easily extend to VM's in a cloud environment. Since using remote memory through a swap device is transparent to VMs, and the proposed system obtains good performance using an RDMA block device based on [74], this may be the best comparison point for FluidMem. Some of Infiniswap's downsides are that it consists of kernel modules for the network block device that require a custom networking stack from Mellanox. These kernel modules are not being kept up to date with the latest 4.x Linux kernel and I was unable to select an upstream kernel that had the necessary prerequisites, but didn't introduce incompatibilities associated found with 4.x kernels. Infiniswap also requires its own custom RDMA memory store that would not integrate with all cloud provider and infrastructures.

Early memory disaggregation papers called for specialized hardware in the form of memory blade servers [58], in part to mitigate network latency. However [58] also suggested partial memory disaggregation as a compromise, with the VM using at least some local memory as a buffer. A hardware architecture for memory disaggregation that makes use of optical switches for fast memory transfers is dReDBox [8] and the authors implement support in Linux using the memory hotplug

mechanism to add additional memory. This mechanism is also used with FluidMem, and it works well when increasing memory capacity. However memory hot-unplug has not been shown to be reliable for decreasing memory capacity.

Beyond pure memory disaggregation, LegoOS [100] is an OS for system resource disaggregation. They implement a memory transfer service, which they note outperforms local swap to RAMdisk because of filesystem buffers that the tmpfs RAMdisk uses. I also noticed this during my testing of a RAMdisk-backed swap. A more valid comparison for FluidMem's case was to look at a DRAM-based block device using the pmem driver [86]. The block device can be configured to avoid the filesystem or page cache on the host operating system. This was the same point of comparison used in [3].

2.2 Dynamically managing VM memory

2.2.1 VM Ballooning

A technique to dynamically balance memory usage among virtual machines is using a balloon driver [111]. This kernel module installed in the guest VM coordinates with the hypervisor to either inflate or deflate its memory allocations within the guest. If memory is needed on the hypervisor, it will instruct the balloon driver to inflate by allocating more pages pinned to physical memory. In response, the guest kernel will register increasing memory pressure, and begin page reclamation, possibly swapping resident pages out to disk. The hypervisor can then allocate the pinned pages to another guest, or for other purposes that it chooses. A significant problem with the ballooning approach is that it takes a relatively long time to reclaim pages that were used within the guest, because they must be flushed to disk before they can be used for other purposes [42].

In the same vein as memory ballooning, Mortar [42] allows dynamic memory allocation and rebalancing of hypervisor memory. While performance improves upon the ballooning approach with intelligent prefetching, cache replacement, and fair share algorithms, it is not a universal solution. It replicates the memcached API for interfacing with applications inside the guest VM, so only

customized applications can make use of it. Like the work presented here, Mortar pools memory in a shared key-value store, but it is only local to the hypervisor.

2.2.2 Hypervisor paging

The commonly cited criticisms of the balloon driver to reduce VM footprints are that it's slow and requires guest cooperation [5]. To address some the weaknesses in the balloon driver, VSwapper proposed in [5] makes modifications to the hypervisor with the visibility necessary to avoid common swap inefficiencies. VSwapper shows the best results when used in tandem with the balloon driver. The design of the VMware ESX hypervisor preferred using the balloon driver under low memory situations, but could also fall back to hypervisor paging to remove memory from the guest [111]. Since hypervisor paging is done without full information of pages used by the guest, the wrong choices for reclaiming pages can be expensive. To make better decisions, [115] looks into how pages are used by the guest and reorders the page reclamation sequence. Rather than swapping pages out to disk, MemFlex [121] moves pages into a shared memory swap device on the host when available. This approach, while useful for performance, goes counter to the idea of memory disaggregation that the VM's memory should be decoupled from the host's memory.

2.3 Live migration

Live migration [17] is a technique for relocating an entire VM, including its memory footprint to another hypervisor. It can be useful for cloud operators to rebalance resource allocations, and prepare for maintenance. While its capabilities include relocating memory over the network, it is not able to resize a VM without a reboot. Performance is only affected temporarily as a machine is stopped on one hypervisor and started again on another. Live migration and memory disaggregation are potentially complementary since live migration is capable of moving execution and memory disaggregation can offload memory from the hypervisor.

The emulation software Qemu [87] provides a different kind of live migration called post-copy migration that makes use of the kernel's userfaultfd mechanism [109]. In this way, a VM can be

migrated before all its memory pages have been copied. When it starts up on the remote host and accesses a page that was left behind, the page fault is caught by userfaultfd and the missing page is retrieved over the network from the previous host. Post-copy live migration is also implemented for containers in CRIU [23]. While both FluidMem and post-copy live migration use userfaultfd, FluidMem differs in that it also has the capability of evicting pages to maintain a constant memory footprint.

Chapter 3

Using RAMCloud for Stateless Network Functions

The concept of disaggregating data storage via user-level network transports and the RAMCloud key-value store was first prototyped for the use case of Stateless Network Functions [44]. Analogous to memory disaggregation with VMs, disaggregating network functions involves separating the state from data storage on a cloud server. The proposal in [44] was to use RAMCloud to remove state from server. The high-performance and low-latency qualities of RAMCloud with Infiniband were used to demonstrate that stateless operation was feasible. To meet the performance targets of the system, it was necessary to improve the design with asynchronous read and write operations to RAMCloud and a simple caching mechanism.

This implementation inspired the modular design of FluidMem's **libexternram** C++ library which allows different key-value store back ends to be linked at compile time to FluidMem's user space page fault handler. The implementation of the StatelessNF storage interface and its optimizations are explained below, verbatim from [44].

3.1 Prototype implementation

Our initial prototype [44] is built using two key technologies: (i) Click [48], as a platform to build new data plane network functions, and (ii) RAMCloud over Infiniband, as a platform which provides a low-latency access to a data store. The central component to our implementation is the StatelessNF client. This StatelessNF client is a class implemented in C++ that carries out data storage and retrieval operations to RAMCloud. It also provides a general data store interface

that can be integrated into Click elements and linked at compile time. We implemented different versions of this interface to explore different designs, detailed in the following subsections. In order to take advantage of RDMA transactions for one-sided writes to RAMCloud and minimal read latency, we used the *infrc* transport mechanism in RAMCloud in preference over the *tcp* transport mechanism that is compatible with an Ethernet fabric. This transport implementation uses the reliable connected (RC) Infiniband transport. The initial connection handshake between a client and a RAMCloud server is done through a socket interface using IP over Infiniband, which encapsulates an IP packet in Infiniband frames. Once a connection has been established between client and server, memory regions are pinned by the Infiniband device driver for DMA transfers bypassing the OS, and further communication is transitioned to the InfniBand verbs interface.

3.2 Blocking read/write

The StatelessNF client provides a read/write interface to a datastore. To minimize changes in Click elements, our first implementation of this functionality is a synchronous (blocking) interface. Typical elements in Click might have a lookup in a data structure (such as a Hashmap). These lookups are performed as part of the sequentially executed code to process the packet. As a local data structure, this is perfectly reasonable. For our first implementation, we simply replace these reads and writes with blocking reads and writes to RAMCloud. That is, the Click element submits a lookup to the storage client interface and does not output the packet until the RAMCloud operation completes, with read data or confrmation of a successful write.

3.3 Async read/write buffer packets

As an improvement, we also implemented an asynchronous (non-blocking) read and write interface to the StatelessNF client (using the asynchronous interface to RAMCloud). To utilize this, the Click element needs to break the packet processing into pre-lookup and post-lookup functionality (or more generally, into processing stages if there are multiple reads, but for simplicity we focus on the single read case), along with integration of the Click scheduler. Consider an example element

which has a push interface, where the push() function extracts some fields in the packet, performs a lookup, and based on the results of the lookup modifies the packet and outputs on a given port of the element. Replacing the lookup with a read to our StatelessNF client requires breaking apart the function. Instead in its push() function, the element extracts fields in the packet to be used for a lookup, submits the read request to the StatelessNF client, and then returns from the push function. We periodically check inside run task(), invoked by the Click scheduler, whether any results have completed. If they have, we call the element's post-processing function, which in turn outputs the packet.

Chapter 4

Architecture

While the dynamic aspects of memory disaggregation hold promise to be a good match for cloud computing, existing works in memory disaggregation have taken shortcuts in its realization. Yet, they have shown promise for the feasibility of big data workloads with adequate performance under disaggregated configurations [37, 38, 90]. However, the other two challenges of memory disaggregation mentioned in Chapter 1, support of full memory disaggregation and flexibility for cloud operators have not been addressed. This chapter will detail an architecture for supporting those two needs without compromising the first challenge, performance. In fact, Chapter 5 will demonstrate that the performance of applications running on FluidMem exceeds that when using another recently proposed alternative for memory disaggregation.

4.1 Performance

The original motivation for extending memory capacity to remote nodes was for increased performance, by supplanting cheaper to disk-based storage with remote memory that can be accessed over high-speed network. While memory disaggregation, and especially full memory disaggregation, brings flexibility to the cloud operator and dynamic capacity to the cloud user, the performance of a memory disaggregation solution must be significant improvement over disk (SSD). The premise of nearly all remote memory paging systems [12, 34, 56, 63, 76, 41, 56, 38] is that their performance exceeds local disk. However, to meet the challenge of near-term improvements in storage technologies, the goal of FluidMem's architecture is to improve on performance of the best swap-based

remote memory architecture, while providing all of the other benefits of full memory disaggregation.

Previous software memory disaggregation prototypes have used the swap interface, common to Linux and most operating systems, to transparently leverage extra capacity [38]. However, while functional to page in and out of remote memory, the swap mechanism was architected for a different purpose than disaggregation.

4.1.1 Swap space is not the same as extra memory

The swap component of the Linux memory subsystem was designed to remove infrequently used pages from DRAM in order to make room for more frequently used pages. The specifics in this discussion are limited to Linux, but the idea applies to all operating systems. User memory on a Linux system can be divided into file-backed pages that have a corresponding location on a filesystem, and anonymous memory without such a mapping. Under low memory conditions, file-backed memory pages, such those in memory-mapped regions created by the *mmap()* system call, can simply be discarded, and those pages that have been modified can be written back to the original filesystem location. There is no need, nor capability, to store these pages in swap space. Anonymous pages on the other hand must be saved somewhere if the system needs to free memory, but the application might access that page again. So, a kswapd kernel thread in the operating system will scan for pages that are considered inactive to either write them back to a file location or add them to swap space. There is an overhead associated with this process in setting and clearing bits marking the page's activity status and causing minor page faults (those that can be resolved without I/O to a swap device).

The above system works well for removing truly non-useful pages from memory, such as memory belonging to system daemons started at boot. Unfortunately, as memory pressure increases, so does the scanning rate and overhead associated with kswapd. In a scenario where remote memory is added for extra capacity, and the working set size is greater than available DRAM, there are hardly any inactive pages to be found. The kswapd process will never be able to catch up, and its overhead is only useful to move the page to remote memory. Thus, *the kswapd overhead is inherent*

in swap-based remote memory.

Other performance limitations with the swap mechanism are tied to its historical role as an interface to rotational hard drives. Recent Linux kernels have improved the swap interface for SSD drives with per-CPU request queues, and by disabling sequential readahead [106]. The current swap overhead was claimed to have been reduced to 4 μs, but in our experiments inside a virtual machine, we observed significantly higher latencies, as shown in Figure 5.1. Additionally, this does not account for the CPU overhead of the kswapd process. Features such as batching and reordering I/O have no benefit for remote memory, which can perform random accesses as quickly as sequential accesses. They will however, increase the latency of an individual page-out request, since the block layer holds on to requests in order to coalesce multiple requests into larger operations.

Concerns about the inherent performance deficiencies of the swap interface were expressed in [95].

> Therefore, once the system resorts to swapping, regardless of how fast or optimized the swap device is (remote memory, etc.), the system performance degrades significantly due to the inherent limitation in kernel swapping method which is designed for very slow devices such as the HDD.

Finally, the swap mechanism is not effective at removing all unused pages. Pages belonging to the kernel are one such category of non-swappable pages. There are also unevictable pages that kswapd will skip over. These are mostly pinned pages by the *mlock()* system call. These pages will add to the minimum footprint a VM will occupy in DRAM as long as it's running. If these pages could be removed from local DRAM, that space could be allocated to pages belonging to running application. There is a potential to increase memory capacity, and performance, by removing these pages from DRAM.

To summarize, limitations of the swap interface with respect to the remote memory use case are:

(1) A latency overhead from minor page faults caused by kswapd scanning

(2) A CPU overhead from kswapd continuously traversing lists of used pages and marking

them as active or inactive.

(3) Some pages cannot be stored in swap space: those belonging to the kernel or those that have been pinned by an application.

4.1.2 An alternative to swap

Even though the swap device is a convenient interface at which to intercept page faults that should go to remote memory, there is an alternative in the Linux kernel, **userfaultfd** [109]. Since Linux 4.3, the userfaultfd feature allows for address ranges to be registered with a file descriptor that can be monitored by a separate user space page fault handler. When a non-mapped address is accessed by the VM in the newly allocated region, the VM vCPU process is suspended, and an event is delivered to the file descriptor. Another process running on the system can monitor the file descriptor, and when notified, perform the tasks of determining what the page should be filled with before resuming the VM. The monitor process can preserve normal page fault handling semantics by returning the copy-on-write zero-page for first time accesses, or the expected data page on a re-access. To control the number of mapped pages in the VM (the amount of space resident in DRAM), the monitor process can choose pages to remove and send to remote memory, fetching them again when there is a re-access page fault. The code needed for eviction, also known as UFFDIO_REMAP was not part of the final userfaultfd patch that was merged into the kernel for 4.3. Since this functionality is necessary for FluidMem, I've kept the originally proposed code by Andrea Arcangeli compatible with the most recent kernel release [108]. I will be submitting this code for inclusion in the upstream kernel.

4.1.3 Order of operations for page fault handling

The sequence of actions that must be taken by the page fault handler for each userfaultfd event are listed below:

(1) Read address of page fault from the userfaultfd file descriptor

(2) Read the page from remote memory

(a) Option A: always read the address from remote memory

(b) Option B: read only when the page hasn't been seen before

(3) If LRU list has reached its footprint maximum:

(a) Evict a page from the VM footprint

(b) Then send the evicted page to remote memory

(4) Return page to VM. If page was read successfully in (2):

(a) Resolve fault by placing data page and waking the VM

(b) Else, resolve fault by placing zero-page and waking the VM

The above list is similar to the sequence for a swap-based remote memory system. The user space page fault handler will make system calls with a privilege switch to kernel space at (1), (3a), and (4). Network transports such as RDMA and DPDK, also operate in user space so they can easily be called from the handler. As an added benefit of remaining in user space, the same data buffer returned from (2) can be used for (4) in a zero-copy read. A zero-copy write can also be performed in (3).

Several opportunities for making operations asynchronous exist. Since FluidMem's performance goal is to be equal or better than swapping to DRAM, every opportunity to reduce latency should be taken. The optimizations completed in the implementation are discussed in Section 5.3.

4.1.4 Scalable memory storage

The choice of a backend storage system has a critical impact on performance. Since the read operation of the page fault sequence shown above must be satisfied before the fault can be resolved, it is always in the critical path of a page fault. Relative to the system calls and execution of user space code, the read and write operations to a backend storage system will dominate latency.

With this criticality of read performance and the unique position of the user space page fault handling code in mind, the attractive architectural choice is to integrate with more than one high-performance key-value backend. In this way a user of FluidMem can choose the best performing

backend for their infrastructure and requirements. A key-value store makes sense as a backend because memory disaggregation involves the replacement of random-access memory with the 64-bit address representing the key and a 4 KB page for the corresponding value.

Since the paging of many applications may be handled by a single backend, it must handle concurrent operations, support low latency reads and writes, scale in capacity and throughput, and provide durable storage. Satisfying all of these requirements in a specialized distributed memory store built into the kernel would be an enormous undertaking. It would be prone to bugs, have security implications for the entire operating system, and be unlikely to be accepted by maintainers of open source kernels such as in Linux.

Looking more broadly at high-performance and scalable data storage systems, there already exist distributed in-memory key-value stores such as RAMCloud [84] and FaRM [27]. These key-value stores are complex distributed systems in their own right, but since they have focused on the intended use case of high-performance in-memory storage in user space, they are more feature complete than the memory stores of previous works. As an example, RAMCloud provides crash-recovery, tolerating node failures without loss of availability to the data store. RAMCloud has latencies of less than 10 μs, for 4 KB reads and writes and is a mature open source software project. At the time of this writing, RAMCloud appears as the best choice for supporting a stable implementation of full memory disaggregation.

4.2 Full memory disaggregation

4.2.1 Transparency

A key hurdle to overcome is how can we transparently intercept remote memory accesses? A natural approach is to hook directly into the page fault handling mechanism of the hypervisor. When an application (the guest OS of the VM running on top of the hypervisor) requests a previously used page which is not resident in DRAM, the page fault is trapped by the hypervisor's kernel. The kernel is free to implement its own mechanism for fetching the page, and once it has

been placed in DRAM, the application can be resumed. However previous kernel space attempts at remote memory [47, 14, 33, 103] have failed to gain acceptance as part of the Linux or FreeBSD kernels, and as a result, have not maintained compatibility with current versions. Furthermore, a kernel space solution can be difficult to maintain and debug.

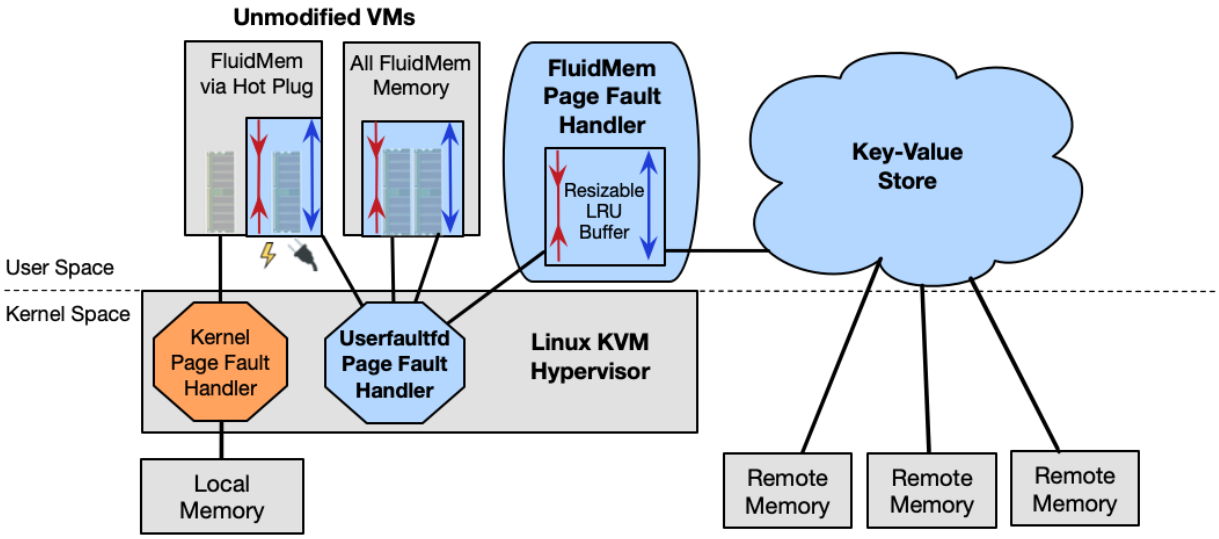
Short of modifying the hypervisor kernel, *userfaultfd* provides another option, where specifically registered memory regions have their page faults resolved by a user space process. In this way, many of the needed facilities for a hypervisor to transparently handle a VM's page faults are already part of Linux and will be maintained going forward. We adapt this user space fault handling mechanism to include functionally to evict pages from the VM into user space on its way to remote memory. The eviction path can be done asynchronously, since removing memory will not interfere with the guest if suitable LRU pages are evicted.

4.2.2 Expanding system memory

Another key aspect of FluidMem's architecture is the ability to dynamically increase a VM's physical memory space without a reboot. As shown in Figure 4.1, two basic modes of using FluidMem exist: the VM could be started with all its memory registered with userfaultfd or the additional FluidMem memory could be added via **memory hotplug**.

An OS such as Linux already has kernel support for memory hotplug to permit addition of new local physical memory during run time. Instead of limiting hotplug to local physical memory, we rather adapt it so that remote memory can also be hotplugged and made visible to the guest OS. In this way, the kernel of the guest OS is given access to more physical page frames and memory pressure is quickly reduced. It is then no longer necessary for the kernel to activate the background page reclamation process, and more CPU cycles are available for applications. This approach of adapting existing hotplug mechanisms to permit the addition (and subtraction) of remote memory also has the virtues that it does not require any modification to the guest OS kernel and only small patches to the hypervisor. All patches made to the hypervisor are being submitted to the kernel developers for upstream inclusion.

Figure 4.1: FluidMem Architecture



The FluidMem page fault handler shown in 4.1 is enforcing the allowable VM footprint size as determined by the LRU list size. For the case of a VM with all FluidMem memory, the list might be sized to the equivalent amount of local memory a VM typically starts with (e.g. 4GB). Page accesses from within the VM should not require an access to remote memory, because they will have a known content of all zeroes. Once the LRU list size has been exceeded, each new page access will also trigger a page eviction of the least recently added page to the buffer. Effectively, the memory usage of the guest will be limited to the number of pages on the LRU list. When a guest VM and re-faults a page back in that was previously evicted, the page will be read from the key-value store and placed at the head of the LRU list.

4.2.3 Shrinking system memory

The LRU list in FluidMem is a data structure maintained in user space by the page fault handler process. A cloud operator can connect to the handler process via a TCP socket and resize the buffer down if they would like to reduce the footprint of the VM without causing a service interruption. A number of pages at the end of the LRU list will be immediately evicted using the userfaultfd interface and be sent to the key-value. Since the size of the LRU list will determine the hit ratio of the VM (local versus remote memory), reducing it too much might cause the VM to slow down, but the VM will continue to run by paging in remote memory on-demand. If the buffer is sized down too much and performance degrades, the operator can increase the LRU list size to return performance to normal.

This approach to managing memory in the guest is similar to hypervisor paging as discussed in section 2.2.2 and shares a drawback. The choice of which page to evict is made by the page fault handler without knowledge of when the page was last accessed. It only knows about the accesses when the page was not present in DRAM. Since the size of the buffer represents the footprint of the VM, the buffer may need to be several gigabytes in size. For large buffer sizes, very little locality information is retained, but the LRU list of pages in the order they were faulted in has been show to provide good performance (see Chapter 5). Optimizations to choose better candidate pages to

evict could improve FluidMem performance in the future.

4.3 Flexibility for cloud operators

4.3.1 Heterogeneous Hardware

Memory disaggregation should not prescribe a particular network infrastructure or backend memory storage systems, but should instead support the heterogeneous mix present in today's data centers. This means that Ethernet and the conventional TCP transport should be supported for moving pages in and out of remote memory, but other transports should be supported as well. For example, those that support RDMA and/or kernel-bypass. This thesis builds the case for a distributed memory storage system in software, but the specific system can vary with cloud provider's preferences or by where engineering expertise has accumulated for specific key-value stores. A cloud provider may opt to even have multiple memory backends supporting different guarantees for memory reliability and performance.

4.3.2 Open source customizations

The largest public cloud providers Amazon and Google have customized their own hypervisor platforms, perhaps for reasons of efficiency, competitive advantages, or infrastructure compatibility. Whatever the reason, it follows that new or existing cloud deployments wishing to deploy memory disaggregation should have similar freedoms. For small institutions, the benefits of having a collaborative open source community to draw upon for innovation is hugely valuable because the institution might not have the engineering resources of their own to make improvements or develop new tools. As new open source key-value stores and network transports that benefit memory disaggregation become available, existing deployments should be able to take advantage of them.

4.3.3 Managed complexity

The number of VMs managed in a cloud data center could be in the tens of thousands. If another service is to be added to the infrastructure, it should afford cloud operators administrative

economies of scale. A key-value store that aggregates extra memory capacity for hundreds or thousands of VMs would be much more desirable than individual hardware components on each server. The ideal infrastructure for supporting FluidMem's architecture would be separate pools of highly reliable systems with large amounts of memory separate from the commodity cloud compute nodes running customer applications. If disaggregated memory for several VM's is concentrated on these key-value stores, it would be critical to ensure that their operation is minimally vulnerable to hardware or software failures. FluidMem's architecture with pluggable key-value back ends would also work on commodity hardware over an Ethernet network. Currently implemented in FluidMem is support for the memcached backend over TCP.

Chapter 5

Analysis

This chapter examines the performance of FluidMem, first starting with raw page fault latency within a VM, then an investigation into code contributions to latency, and finally full application performance is demonstrated with the Graph500 benchmark and the cloud document store MongoDB.

5.1 Test platform configuration

These experiments were run on a cluster of dual-processor Intel Xeon E5-2620 v4 servers running the Linux 4.20 rc7 kernel and CentOS 7.1 distribution. An Infiniband network connected the nodes with Mellanox ConnectX-3 FDR (56 Gb/s) cards. The Infiniband network was used for the NVMe over Fabrics (NVMeoF) tests using swap and FluidMem with RAMCloud tests. For the FluidMem tests with the memcached backend, the IPoIB interface on the Infiniband cards was used. RAMCloud was configured to provide 25 GB of storage and was run on a different server than the one running the test VM. The replication feature with RAMCloud was not turned on. The RAMCloud server process was restricted to NUMA node 0 of the server, which is the node the Infiniband interface connects to.

The kernel settings were left at the default, except transparent huge pages were tuned off, cores dedicated to running the experiments were isolated, and the Infiniband card's Max Read Request was set to 4096,

5.1.1 Infiniswap comparison

A swap device backed by remote DRAM using NVMe over Fabrics [79] is used for comparing FluidMem to Infiniswap [38]. The NVMe over Fabrics project has superseded the RDMA NBDx block device [74] which Infiniswap uses. Using this backing medium as a substitute for Infiniswap serves as an optimized case without the complexity of the distributed block management functionality found in Infiniswap. The remote storage protocol of [38] involved synchronous writes to the RDMA block device, and asynchronous rates to a local disk drive. A synchronous write to the NVMe over Fabrics block device using O_DIRECT semantics should be comparable to a write to an Infiniswap block device. Reads from swap space must be synchronous in both cases.

The NVMe over Fabrics device is made available on the hypervisor by loading a kernel module and connecting to the target via RDMA. The libvirt configuration for the KVM VM presents the device as /dev/vdb in the VM for a swap device. Since multi-queue support is part of the NVMe over Fabrics specification, blk-mq is enabled with the 4.20 rc7 Linux kernel.

Kernel modules for the NVMe over Fabrics storage target were loaded on a different server accessible via an FDR (56 Gb/s) Infiniband link. The target was a 10 GB allocation of DRAM accessed via the /dev/pmem0 interface [86].

5.2 Latency micro-benchmarks

To better understand the raw page fault latency seen by different FluidMem backends and how they influenced application performance in 5.4, we measured access latencies using pmbench [116] from within a VM. For the swap cases, the VM running the benchmark had 1 GB of local memory and a 20 GB swap device on different mediums (DRAM, NVMe over Fabrics, and SSD). The FluidMem VM was also sized as 1 GB but it was registered with the FluidMem page fault handler on boot. It was allowed a 1 GB LRU list in FluidMem, so up to 1 GB of pages could reside in DRAM before any pages were evicted. An additional 4 GB of hotplug memory was added, raising the capacity to 4 GB, but maintaining the 1 GB LRU list. The benchmark was run with a working

set size (WSS) of 4 GB, that it touched first to fault all pages once, and the ran for 100 seconds, recording latencies of 4 KB read and write operations. The VM footprint was restricted to 1 GB in both swap and FluidMem cases, while the WSS of pmbench was 4 GB. All tests were run on the same VM in the same Linux 4.20 rc7 kernel patched to disable asynchronous page faults only for userfaultfd regions.

The average latencies are shown in 5.1 and the histogram of sample counts at various latencies are plotted in 5.2.

The results shown in Figure 5.1 indicate that the lowest page fault latencies are with FluidMem and are about equal between the DRAM key-value implementation and the RAMCloud key-value implementation. This shows that the latency hiding operations done for the RAMCloud backend were effective.

Figure 5.2 shows histograms from the same data used in calculating average latencies in 5.1. Both axes are on a log scale. All graphs have characteristic peaks below 1 μs and a second peak close to the average given above. The second peak varies between <5 and 100 μs. The first peak is for main memory accesses where the second peak corresponds to faults that went through a page fault handler (either userfaultfd or swap). Some observations from the plots are that reads are generally faster than writes for the FluidMem cases, but the distinction is less clear for swap. All the configurations have a long tail after the peak caused by page faults. This appears to be a result of the medium because the tail is consistent between FluidMem backed by DRAM (Figure 5.2a) and swap backed by DRAM (Figure 5.2b).

5.3 FluidMem optimizations

In Figure 5.3, we see the distribution of latencies for individual components of FluidMem with the code paths used in a first time access highlighted. These measurements were taken with FluidMem running on a bare-metal server with a test application for generating page faults. There is no virtualization overhead involved. The code was instrumented to time the contribution to overall page fault latency for noteworthy sections of the code. For timing purposes, the asynchronous

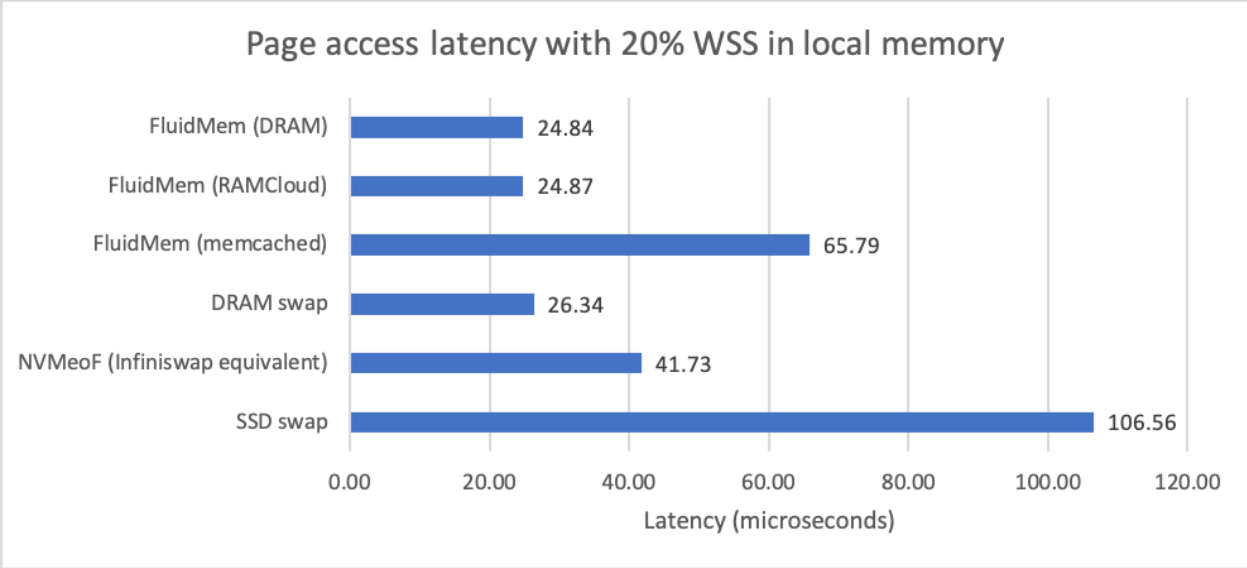
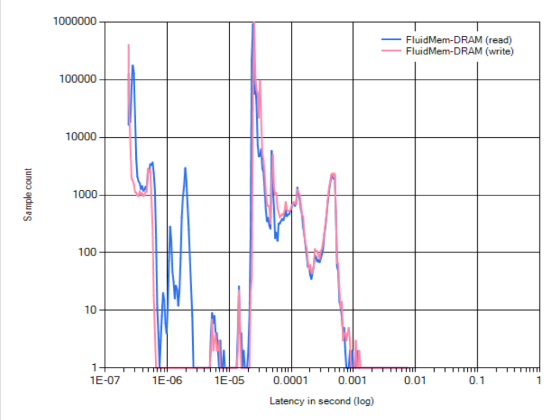
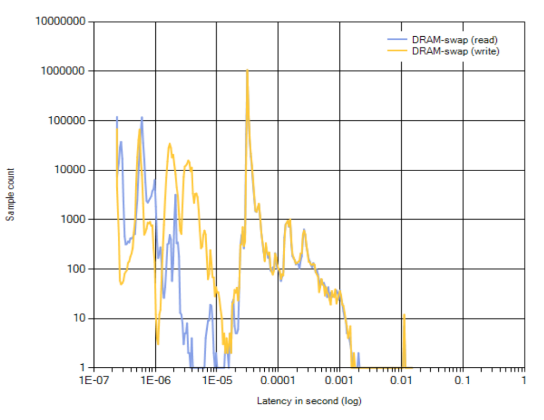


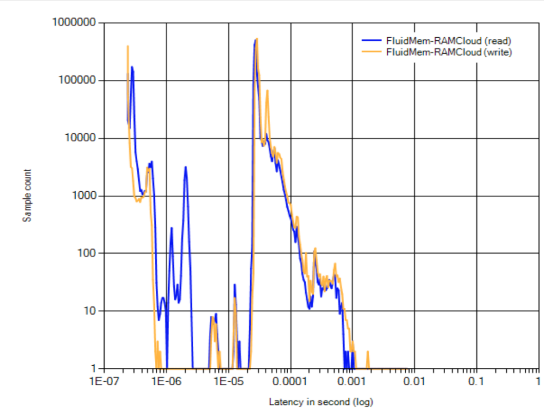
Figure 5.1: Page access latencies with 1 GB local memory and 4 GB pmbench working set size



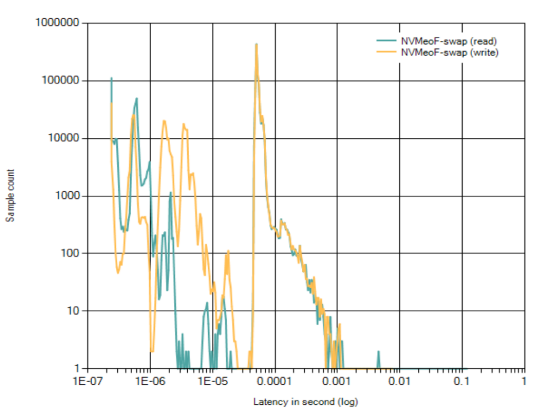
(a) FluidMem with DRAM (24.84 μs avg.)



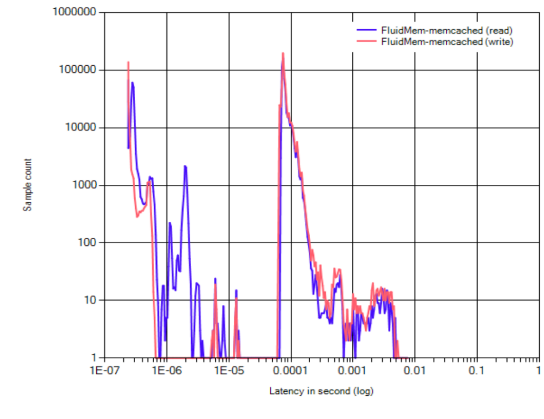
(b) Swap with DRAM (26.34 μs avg.)



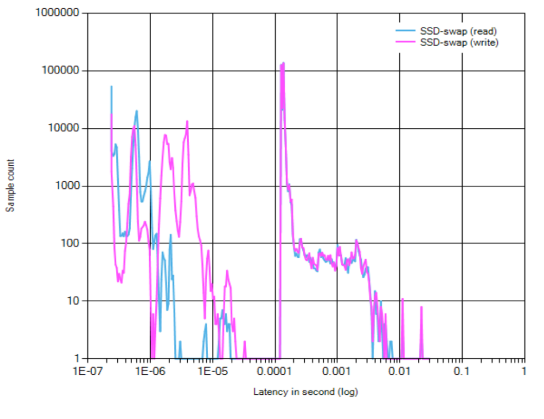
(c) FluidMem with RAMCloud (24.87 μs avg.)



(d) Swap with NVMe over Fabrics (41.73 μs avg.)



(e) FluidMem with memcached (65.79 μs avg.)



(f) Swap latencies with SSD (106.56 μs avg.)

Figure 5.2: Histograms of pmbench latencies for FluidMem vs. Swap with different backends

prefetch and eviction code was disabled. The timing latencies in the green box to the left relate

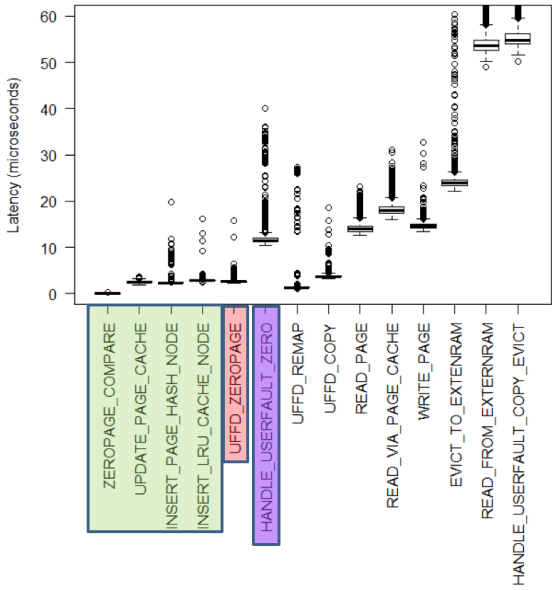


Figure 5.3: Latencies of FluidMem code involved the first time a page is accessed

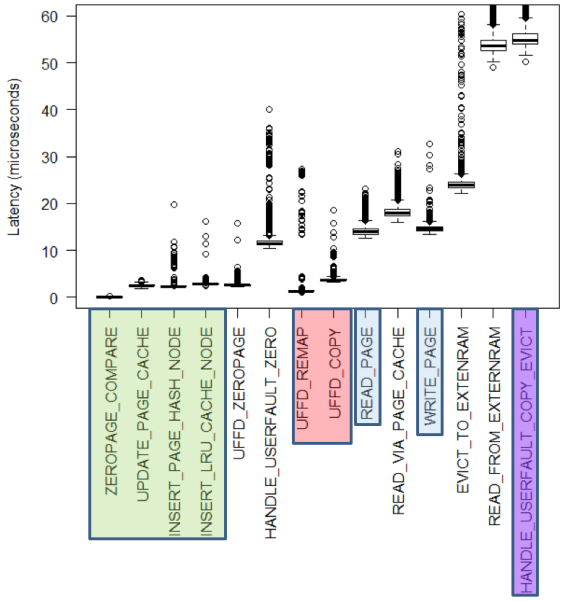


Figure 5.4: Latencies of FluidMem code involved in a page re-access (from remote memory)

to the cache management of internal data structures, which add up to about 7 μs. The red box is for the userfaultfd system call that creates a special copy-on-write zero-filled page for the faulting process with a total delay of about 4 μs

Figure 5.4 shows the code paths that are active during a page re-access. There is still 7 μs of overhead for cache management. During re-access faults, two userfaultfd system calls are required, UFFDIO_COPY to insert the page that was received from the key-value store into the faulting process, and UFFDIO_REMAP to evict a page from the process for sending to the key-value store. Additionally, there are two remote memory functions activated, one to read the page and the other to write it. The total latency for this code path is about 56 μs. Note that this is higher than figures mentioned elsewhere in this thesis because all asynchronous operations were turned off for this test.

5.4 Application Use Cases

This section describes use cases of standard applications running on FluidMem, demonstrating that expanding a VM's physical memory capacity allows unmodified applications to improve performance.

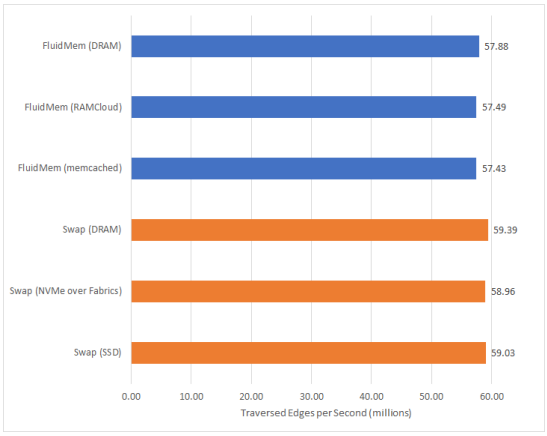
Applications that will benefit the most from FluidMem are those that load large datasets or indexes stored in memory. With FluidMem, the amount of physical DRAM on a server is no longer the upper bound for the amount of addressable physical memory, so the application's working set size is free to grow beyond local DRAM and spill into remote memory. Some applications can easily partition their working set into discrete chunks that will fit into the RAM available on each node and spread across multiple nodes. Such applications are not likely to see benefit from adding more memory. However, if a partitioning method is not known before hand, then FluidMem can improve performance by allowing more of the dataset to be resident in RAM. While partitioning of the data for distributed operation may be possible with explicit refactoring of the application's code, FluidMem provides an alternative solution to load these datasets into memory without requiring development effort from system engineers.

5.4.1 Graph500

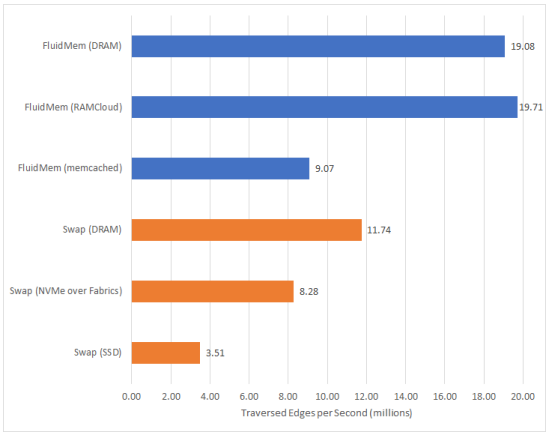
The Graph500 benchmark was chosen to evaluate how page fault latency on various FluidMem backends affected overall application performance in a VM capable of full memory disaggregation. Completing a breadth-first search (BFS) traversal is generally a memory-bound task due to irregular memory accesses [11]. For this reason we chose the sequential reference implementation of the Graph500 benchmark [73], which is a BFS graph traversal. We note that there are many prior works on parallel BFS implementations on distributed clusters [11], but the focus of this thesis is on full memory disaggregation in the cloud, limited to individual VMs, not a single distributed shared address space. So optimizations that would involve distributed parallelism are not considered.

All experiments were run on a KVM virtual machine with 2 vCPUs and 1 GB of local memory resident on the hypervisor. For the swap configuration, this also meant a memory capacity of 1 GB for the VM. Block devices backed by the different stores were configured as swap space within the VM. The libvirt configuration to present the block device to the VM used the virtio driver with caching mode set to "none", meaning O_DIRECT semantics were used and the host's page cache is not involved. This setting was critical for an accurate comparison between swap in FluidMem. With the caching mode set to "witeback", writes to the swap device would be buffered in the hypervisor's cache. Using "witeback" mode actually made swapping to DRAM slower because of the extra caching layer. For FluidMem, the LRU list was set to 1 GB and an additional 4 GB of hotplug memory was added without increasing the LRU list size. Swap was turned off for FluidMem tests. The FluidMem page fault handler ran on 4 cores sharing the same NUMA node as the 2 vCPUs, and closest to the Infiniband interface.

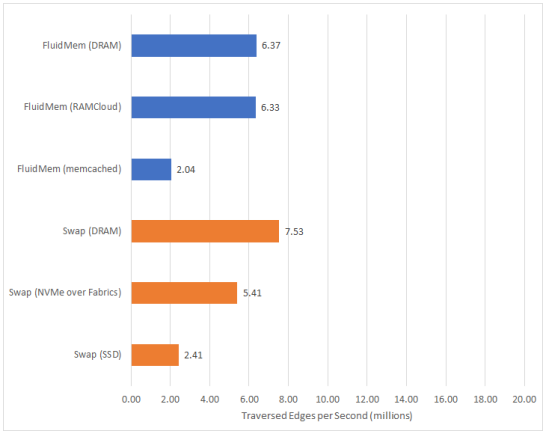
Figure 5.5 shows the results of Graph500 run at scale factors 20-23 for three backends in FluidMem and swap configurations. The metric used is (millions of) traversed edges per second (TEPS). The benchmark would create the graph in memory and then time 64 consecutive BFS traversals, reporting the harmonic mean TEPS and standard deviation. The harmonic mean measure is reported in Figure 5.5. The working set size is altered by changing the scale factor of the



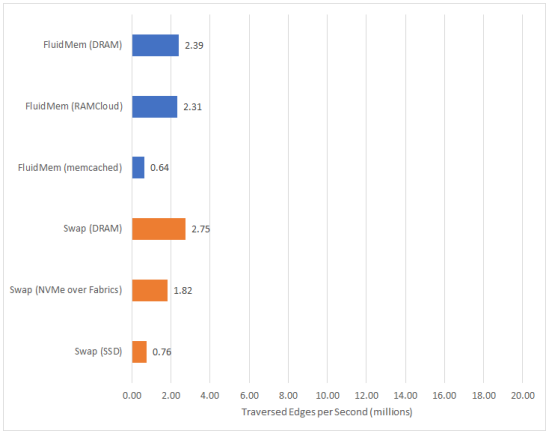
(a) Scale factor 20 (uses 60% local mem)



(b) Scale factor 21 (uses 120% local mem)



(c) Scale factor 22 (uses 240% local mem)



(d) Scale factor 23 (uses 480% local mem)

Figure 5.5: Graph500 performance for scale factors 20-23

benchmark, which changes the graph size. A scale factor of 20 constructs a graph that fits inside the 1 GB footprint and involves no page faults through swap or FluidMem. The largest scale factor tested of 23 corresponds to a graph size of about 5 GB.

Figure 5.5a shows the harmonic mean TEPS of scale factor 20. Since this test involved only local accesses, it was used to assess the overhead of FluidMem's full memory disaggregation. Since FluidMem traps to user space and performs a check for each page it hasn't seen before, the cost of a minor page fault for the FluidMem case is higher. For all cases, the number of minor page faults was about 152,700. The average slowdown of FluidMem backends versus swap for 64 iterations at scale factor 20 was 2.6

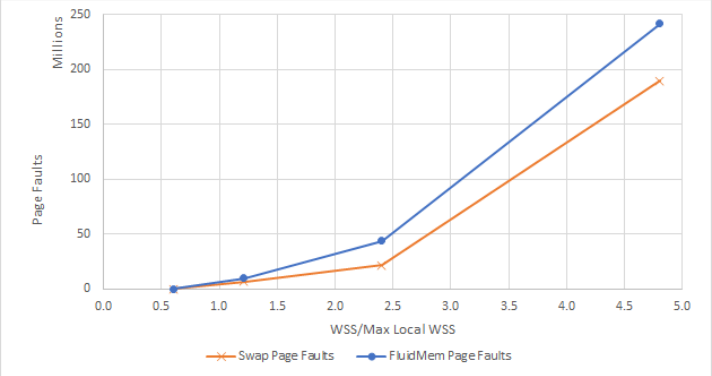


Figure 5.6: Graph500 page faults for scale factors 20-23

The FluidMem configurations do significantly better than swap-based configurations at scale factor 21 (WSS 120% of local memory) shown in Figure 5.5b. The large performance difference is primarily because FluidMem allows more unused pages to be removed from DRAM as explained in Section 4.1.1. FluidMem can fill its entire 1GB of local memory with application pages. To test this hypothesis regarding a benefit near the local memory to remote memory threshold, the number of page faults at each scale factor are plotted in Figure 5.6. The results indicate that FluidMem actually had 3,000,000 more page faults than swap. However, FluidMem had 577,000 fewer voluntary context switches than swap. Voluntary context switches correspond very closely to the number of minor page faults with FluidMem, which indicate the number of times the VM was paused while waiting for the user space page fault handler. So, taken together approximately 2,500,000 page faults in FluidMem did not register with the guest kernel. This is likely because those page faults involved kernel pages which swap space is not capable of placing in remote memory.

Another aspect of Figure 5.5b that has promise for cloud data centers with standard Ethernet networks is that the memcached backend for FluidMem performs better than swap backed by NVMe over Fabrics and SSD. This is due to the increased amount of application pages in local memory as discussed above, but the raw latency of FluidMem with memcached is still faster than swap to SSD (Figure 5.1).

At scale factor 22, FluidMem with RAMCloud still outperforms swap with NVMe over Fab-

rics, but the results for DRAM backed storage are inverted (swap becomes faster than FluidMem). Given that the raw page fault latencies for FluidMem backed by DRAM are lower than swap backed by DRAM, a possible explanation is that the kswapd process within the guest was better at choosing pages to swap out than FluidMem's LRU was at choosing pages to evict. This is supported by the observation that FluidMem incurs more page faults than swap at the same scale factor (shown in Figure 5.6).

The same trend persists at scale factor 23 (480% WSS of local memory) where FluidMem with RAMCloud outperforms swap backed by NVMe over Fabrics. At even higher scale factors, it is expected that the relative performances will be approximately the same as scale factor 23.

Even with the working set size as a much greater percentage of total memory, applications should still run to completion. Infiniswap [38] only explored applications with 50% of their working set in memory and cited problems with thrashing and failing to complete beyond that split of remote memory. We have demonstrated FluidMem memory disaggregation completing the Graph500 benchmark with only 21% of the working set resident in local memory.

5.4.2 MongoDB

Not all applications are able to take advantage of extra memory through the swap interface. Since many cloud applications are run by users who did not originally write the code, or the application may be close sourced, a system for full memory disaggregation in the cloud should benefit even unmodified applications that are unaware of the presence of remote memory.

MongoDB is a document store commonly used for cloud applications that facilitates fast retrieval of data stored on disk by caching its working set of documents in memory. MongoDB has two storage engine options, mmapv1 that uses the memory-mapping system call *mmap()* to let the kernel manage which pages are cached in memory, and a newer engine WiredTiger that uses a combination of an application-specific cache and the kernel's filesystem cache. While future MongoDB versions will deprecate the mmapv1 storage engine, it is an example of a wide range of applications that use mmap() [68, 60, 104, 85].

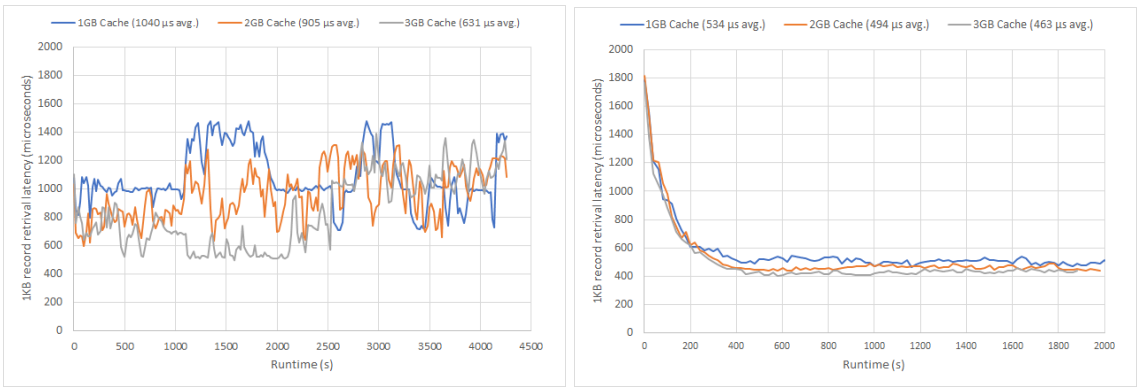
As discussed in Section 4.1.1 one of the limitations of swap space is that it cannot store memory-mapped pages. When an application that makes use of mmap() is run with remote memory provided by swap space, the performance is the same as without remote memory because the operating system will write out pages from the memory mapping to disk, not to remote memory through swap space.

The workload chosen is from the Yahoo Cloud Serving Benchmark (YCSB) suite [20], consisting of of read requests of 1 KB records. In this way, data is initially read from disk, but then held in the memory until read again or evicted to making room for more recently read records. The MongoDB server is run on a VM with 1 GB of local RAM for the swap case. The swap device is backed by either DRAM via /dev/pmem0, a NVMe over Fabrics target device in DRAM on another server, or a local SSD partition. For the FluidMem case, the VM had 4 GB of RAM capacity, but it was limited to a local footprint of 1 GB set by the LRU list. Swap space was turned off for tests with FluidMem. Configuration and tuning recommendations from the Mongo DB Production Notes [70] were applied: transparent huge pages were disabled, NUMA was disabled, vm.swappiness was set to 1 for mmapv1 tests and 100 for remote swap tests with WiredTiger, disk readahead was set to 32 for mmapv1 and 0 for WiredTiger.

YCSB workload C was run from the MongoDB VM to reduce the impact of network latency in the results. Since the use case of full memory disaggregation most directly applies to a single server, the evaluated MongoDB configuration was not sharded across several servers. A VM with 3 vCPUs, all on NUMA node 0 of the hypervisor was used, where the kernel was free to schedule the MongoDB and YCSB processes on any of the vCPUs. Before initiating the measured experiment, the dataset records were inserted into the MongoDB store by YCSB. The dataset occupied approximately 5 GB on a local SSD available within the VM using the virtio driver with no caching. To ensure that the MongoDB cache and kernel page cache had been flushed, the VM was rebooted between tests. The cache size of the WiredTiger storage engine was varied between 1 and 3 GB.

Latency results for the WiredTiger storage engine on both swap backed by NVMe over Fabrics and FluidMem backed by RAMCloud are shown in Figures 5.7b and 5.7b, respectively.

While average latency decreases for both remote memory configurations with increasing cache size, the WiredTiger storage engine is unable to smoothly take advantage of swap space as extra capacity for the workload. Regardless of the cache size configured, the storage engine has difficulty balancing pressure from kswapd swapping memory out with the pages that will be needed for future requests. The latency numbers represent the time it takes to read a 1 KB record from the MongoDB dataset on disk. Some of the records will require disk I/O, while others can be read from the in-memory cache. Only FluidMem is able to give the storage engine the overhead it needs to hold a sizable cache of records in memory.



(a) Read latency with WiredTiger on swap (NVMe over Fabrics)

(b) Read latency with WiredTiger on FluidMem (RAMCloud)

Figure 5.7: MongoDB read latency with WiredTiger storage engine

5.5 Minimizing VM Footprint

Figure 5.1 demonstrates FluidMem's ability to perform full memory disaggregation. Without FluidMem enforcing the LRU list size, a VM will consume 317 MB of memory just from booting to a command prompt. KVM's balloon driver for reclaiming guest pages reaches its maximum size when the VM footprint is still 64 MB. When FluidMem's LRU list is reduced, a VM with a DRAM footprint of 180 pages is still able to accept SSH connections. At 80 pages, The VM still responds to an ICMP echo request every 1 s. In order to reduce the footprint to 1 page, full virtualization using Qemu [87] was used to avoid a deadlock observed with KVM hardware-assisted virtualization.

See Section 6.7 for a discussion on the need for full-virtualization at low VM footprints.

	VM footprint (pages)	VM footprint (MB)	Response to SSH	Response to ICMP	Revived by increasing footprint
After startup	81042	316.570	Yes	Yes	N/A
Max VM balloon size	20480	64.750	Yes	Yes	N/A
FluidMem (KVM)	180	0.703	Yes	Yes	Yes
FluidMem (KVM)	80	0.300	No	Yes	Yes
FluidMem (full virtualization)	1	0.004	No	No	Yes

Table 5.1: Summary of the effects of reducing VM footprint to 1 page

Chapter 6

Discussion

6.1 A more intuitive interface

Simply as a result of the swap interface's historical role in providing slow disk-based storage, there is a hurdle for user and developer education if swap space were to be used for providing remote memory. Many high-performance computing systems have adopted a diskless architecture for energy efficiency [93], cooling capacity, reduced complexity, and cost. These systems have access to high-speed parallel file systems, and otherwise intend to complete all of their work in DRAM, so there is very little payoff for having swap space. Requiring swap space for a model of memory disaggregation would be a large shift for these environments.

The entrenchment of this skeptical approach to swap space extends to cloud computing. Many of the default cloud computing instances do not provide an additional disk volume for swap space. If a user is sure they want swap space, the recommendation is to create a loopback image to use as a swap file, or pay for an additional storage volume.

The addition of new memory via the hotplug interface alerts the kernel of additional free page frames that it can manage. These page frames can transparently be used by the kernel to satisfy new memory allocation requests. Furthermore, hotplug memory integrates with the existing NUMA model of defining zones with different relative distances between them. If hotplug memory is added on a different virtual NUMA node than the one on which local vCPUs are located, then the kernel will recognize that the CPU should use local memory first before hotplugged remote memory. Since libvirt version 3.10.0, it is possible to define a virtual machine configuration with

preconfigured distances between NUMA nodes [57]. A possible strategy to taking advantage of this feature would be to preconfigure VMs with unfilled NUMA nodes at varying distances and then hotplug FluidMem memory to the NUMA node best matching the remote memory's latency.

For these reasons, in addition to the performance, full memory disaggregation, and flexibility benefits, FluidMem's model of adding additional physical memory to the VM will be easier for users and cloud operators to adopt than existing swap-based proposals.

6.2 Genome assembly

Scientific applications working with very large datasets such as *de novo* genome sequencing applications often struggle to find machines with enough memory to run the desired analysis. Being able to use every last page of DRAM for the application without the overhead of less-used operating system pages is likely to have performance benefits similar to what was demonstrated with Graph500 at scale factor 21 (Section 5.4.1). This section demonstrates the feasibility of running genome sequencing applications SOAPdenovo [61] and Velvet [119] on FluidMem.

In genomics, the identification of a contiguous sequence of DNA base pairs where no reference genome is available is called *de novo* sequencing. Its applications include sequencing new genomes or associating DNA material in a sample with specific organisms (metagenomics). In contrast to other scientific fields that have implemented modeling or analytic codes in a way suitable for parallelization and using the combined resources of many nodes, the problem of assembling a genome sequence is commonly done using large graph structures stored in memory. High-throughput sequencers read DNA samples in very short segments, a few hundred base pairs or less, where overlaps between segments form the basis for identifying a contiguous sequence (contigs). A modern computational technique for matching reads and identifying contigs is constructing a de Bruijn graph with vertices representing the suffixes of all reads and edges where a prefix matches a suffix. An assembler is then able identify candidate genomes by finding Eulerian cycles [19]. For a complete human genome, the construction of the de Bruijn graph using the memory-efficient SOAPdenovo assembler [61], even after error reducing preprocessing steps took 140 GB of RAM [55]. Other

assemblers would take terabytes of memory for a human-sized genome [97]. Since the analysis of a de Bruijn graph is not easily parallelizable [97] and the traversal does not exhibit significant spatial locality, the approach used by most assemblers is to construct and hold the entire graph in memory. Thus the maximum genome size that can be sequenced is typically limited by the amount of RAM on a single node. While it should theoretically be possible to run the assembly to completion by spilling over to swap, the next section will show how it would be much more desirable for the genome assembly to complete with FluidMem on RAMCloud rather than SSD swap.

Using FluidMem, two genome sequencing applications SOAPdenovo [61] and Velvet [119] are able to complete assembling the human chromosome 14 from the GAGE dataset [94] on a VM with 72 GB of memory (60 GB local, 12 GB FluidMem). Testbed nodes for the evaluation only have 64 GB of DRAM each, so the only other way of completing the assembly without FluidMem was to combine the DIMMs from two machines for a node with 128 GB of local DRAM.

6.3 Memory as a Service

The architecture described in this thesis meets the challenges of memory disaggregation in the cloud, but it also opens up the path for integration of FluidMem as part of a Memory as a Service capability. The core needs of performance, bi-directional resizing, and flexibility have already been covered in previous chapters, so the focus here is on the additional requirements of Memory as a Service and how the FluidMem architecture can meet those needs.

6.3.1 User and administrative APIs

As shown in the proposed FluidMem API supporting Memory as a Service in Figure 6.1, the desired API calls differ between a cloud customer and a cloud administrator. The customer should only be able to add to or remove memory from their own VMs. However, they should be separated from the hypervisor interface that typically carries out hotplug actions. The customer API endpoint would only have enough privileges to invoke the libvirt hotplug and hotremove functions.

On the other hand, the cloud administrator would need to perform tasks around managing

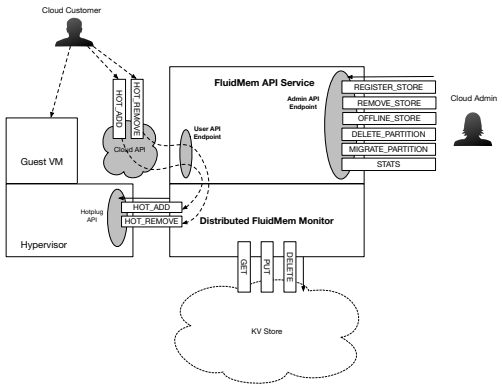


Figure 6.1: FluidMem Memory as a Service Architecture

key-value storage backends. They may want to initiate the live migration of a VM, where the VM's partition in backend storage would need to be re-associated with another hypervisor. The administrator may also want to add or remove additional backend stores or modify available capacity. These are potentially disruptive actions, so the authentication of a user with administrative privileges must be ensured by the API.

The OpenStack cloud management software is not shown in the diagram, but it may be a good place to implement the API since it already supports authentication for multi-tenant environments and already contains metadata on each VM.

6.3.2 Service Isolation

A public cloud is almost certain to be a multi-tenant environment, where multiple users can be assigned VMs that are co-located on the same physical server. Users should feel sufficiently isolated from each other, such that the activity of one customer's VM does not adversely affect the performance of another VM. With respect to Memory as a Service, this means that the hypervisor should be able to handle heavy paging activity from one VM with sufficient overhead should other VMs on the same hypervisor begin intensive paging as well. The acceptable oversubscription ratio of maximum memory traffic from all VMs to the remote memory paging capacity of the hypervisor will be specific to a cloud and the guarantees it provides customers. However, all Memory as a

Service implementations should be able to scale aggregate handling capacity per hypervisor with the number of VMs running.

Similar to performance isolation, Memory as a Service should also provide isolation for security purposes. A malicious user of one VM should not be able to snoop on or redirect the remote memory traffic of another VM. Additionally, listings of resources shown to one tenant should not divulge information about other tenants using the memory service. Since the administrative functions of FluidMem are carried out on the hypervisor and are unavailable to the guest, the vulnerability of FluidMem is limited to traditional hypervisor attack vectors, and the cloud user API which would limit the actions available to each authenticated user.

6.4 Optimizations

A motivation described in Section 4.3.2 for making FluidMem open source was to benefit from the innovation of other open source projects and welcome the contributions of individual users. In the near future, other key-value stores may emerge that have better performance for the memory disaggregation use case than RAMCloud. Since 4 KB is relatively large message size for the Infiniband messaging transport used by RAMCloud, the overhead of registering memory buffers in advance and using RDMA may be a good trade-off. A key-value store supporting zero-copy operations from the buffers in the page fault handler to the remote key-value store may further improve performance.

Other possible optimizations are compressing memory pages as they are being sent over the network and automatically scaling the LRU list according to a VMs working set size (see Section 6.8). A key-value store that encrypts memory data during the transfer over the network, and on disk, could be beneficial to use cases handling sensitive data.

The prefetch mechanism that was implemented in FluidMem had a benefit for certain synthetic workloads, but it was not adaptive enough for the general case with unpredictable workloads. The ability to turn on and off prefetch and improved access pattern detection algorithms are other areas to explore.

6.5 Fault tolerance

By virtue of RAMCloud's replication of all data that gets stored, FluidMem supports transparently handling failures of remote memory nodes. Before RAMCloud returns a successful write to its client (i.e. the FluidMem page fault handler), the data is be stored in the DRAM of a second server (if configured) [84]. As long as all replicas don't fail at the same time, the data will remain available. Nodes that are not dedicated as the master node for a particular partition will periodically flush the key-value pair for which they are a backup for to disk. If the master node fails, a subsequent request for that key will require the secondary server to fetch the data off disk, but the request will still succeed. RAMCloud also has mechanisms to recover from the crash by redistributing data among other nodes so that accesses are served by DRAM again [80].

For the case of memory disaggregation, the aspect of availability is critical. Persistence is only relevant for cases like RAMCloud's crash recovery. The remote memory should be online and available at all times a VM is running. If the VM or hypervisor crashes, there will be no need to recover the data.

The edge case that challenges RAMCloud's high-availability is an event where all servers containing a replica go off-line at the same time. Such a network-wide outage is not a vulnerability unique to FluidMem. To mitigate data availability concerns, the cloud operator should configure RAMCloud with replicas in different failure domains (i.e. different power sources and links to different network switches).

6.6 Planning for persistent memory

Despite the upcoming availability of persistent memory storage in DIMM form-factor modules (NVDIMMs) [83], the need for memory disaggregation will continue. Additionally, there will be advantages to centralizing installations of new persistent memory and making them available to VMs through a system like FluidMem. As discussed in Chapter 1, the need for extra memory capacity has been around since at least the 1980s. It's unlikely that this will change in a short

matter of time. New persistent memory devices such as Intel's Optane DC [83] cannot be retrofitted to old systems since they require extensions to the processor architecture. Additionally, the model for making persistent memory available in Linux is different than conventional DRAM. It will be made accessible to the kernel while carrying the special designation of ZONE_DEVICE [122]. This special type of memory address space can be accessed by applications explicitly coded to use it or as a block device, but it is not available for general memory allocations. As this thesis has already discussed, modifications to cloud applications are not always feasible, so persistent memory as it is proposed today would not benefit legacy cloud applications. The block device option is not attractive for VMs either, because the swap interface is not any faster than FluidMem's page fault handler and does not provide full memory disaggregation.

Instead, FluidMem's architecture is already positioned to integrate persistent memory as backing storage for a key-value store such as RAMCloud. The increased capacity of NVDIMMs would increase the pool of memory available to cloud VMs, and the lower cost would benefit cloud operators.

6.7 Memory disaggregation with hardware virtualization

When initially evaluating the full memory disaggregation use case with FluidMem, KVM hardware-virtualized VMs could not operate with a footprint less than on the order of 1000 pages. However, a VM that was fully virtualized in software using Qemu [87] would continue to run with a footprint of just one page. The issue was the assumption by KVM's hardware virtualization that operating system pages would not be removed from memory regions (as done by userfaultfd). Asynchronous page faults, which allow other vCPU threads to continue after a single vCPU stops with a page fault, were occurring in sections of code that were supposed to be atomic. The fix was a kernel patch to disable asynchronous page faults only for userfaultfd memory regions, but for some applications this could be a slight performance handicap. A special case handling asynchronous page faults in KVM was added to the kernel for pages that are swapped out. It should be possible to use this mechanism for userfaultfd regions as well, but this code change to the kernel is the

subject for future work.

Running hardware-virtualized VMs also required a code update to UFFDIO_REMAP. Since hardware-virtualized VMs have page tables of their own (the guest page tables), they must be updated when mappings in the hypervisor's page table change. Examples are swapping out a page and removing a page via UFFDIO_REMAP. The API to make this notification was updated between the initial coding of UFFDIO_REMAP for Linux 4.3 and Linux 4.13. Once UFFDIO_REMAP was updated to use the new API, evicting pages from the VM no longer caused a kernel panic

6.8 Sizing VM's by working set size

With current cloud computing models, the penalty for under sizing a VM can be large. An application may fail to start, or even waste resources and fail in the middle of its run. So the incentive for the user is to overestimate memory capacity. The user pays in cloud computing costs for this overestimation, and cloud operators lose out on efficient server consolidation [77]. Full memory disaggregation as proposed by this thesis would allow the capability of downsizing a VM's footprint. Starting from the hypothesis that most general-purpose virtual machines are over-provisioned, it follows that marginally decreasing the footprint will not have an effect on performance. To find the optimal VM size, a common technique is to construct a Miss Ratio Curve to deduce the VMs working set size [112], which the VM size should match. However, existing non-intrusive techniques are unable to construct a MRC for sizes below the VM's current allocation [77]. FluidMem's downsizing capability could be used to reduce the footprint, measure resulting changes in the eviction rate, and construct an MRC.

Efficient techniques for constructing a VM's MRC have been proposed. Geiger [113] is a system that passively monitors evictions from a VM to swap space using a ghost buffer. The ghost buffer is analogous to FluidMem's LRU list. Rather than just evicting the LRU candidate as with FluidMem, Geiger records distances between eviction and page reuse to estimate a MRC. One of its limitations is that it has no visibility into situations where the working set size is smaller than VM size. By implementing Geiger in the user space page fault handler, and using FluidMem to

reduce the VM footprint, this limitation would be removed. The working set size of a VM could be estimated despite initial sizing.

The calculated working set size with full memory disaggregation would actually be less than with a conventional VM, because it would not include infrequently used pages that belong to the operating system. Previously, the resident set size of the VM included all operating system pages plus the working set size of the application. Since the hypervisor cannot distinguish between guest operating system pages and application pages, the working set size of the VM was assumed to be equal to the resident set size. However, FluidMem would be able to reveal the true size of the VM's working set size.

6.9 Memory disaggregation with containers

The scope of this thesis was limited to the context of virtualized cloud data centers and an architecture for providing transparent full memory disaggregation was presented. However, containers are becoming more popular for running user applications in the data center with platforms like Docker [25]. While VMs will continue to fill useful purposes for use cases that require strong isolation from other cloud tenants, the performance benefit of running a container on bare-metal hardware will appeal to some customers. Besides performance, if FluidMem were adapted to perform memory disaggregation for containers, operators of container clouds would benefit from the flexibility of using different memory backends to consolidate memory storage. Hybrid clouds with VMs and containers would benefit from a unified platform for memory disaggregation.

In a VM, the translation between a guest physical address and a host virtual address (where userfaultfd operates) provides a convenient place to intercept page faults and inject pages fetched from remote memory. Since FluidMem modifies Qemu to register the guest physical address space with userfaultfd, no modifications to the guest were required.

When a container starts, there is not a specific new memory allocation for the container, so not a transparent opportunity to register with userfaultfd. Page faults in a container are handled natively by the kernel without any translation to virtual addresses. Without the virtualization

layer, a native page fault to DRAM will be handled faster than the 25 μs range measured in the VM in Chapter 5, putting more pressure on the performance of the page fault handler, and possibly requiring more work to optimize userfaultfd system calls.

There are two approaches to transparently incorporating userfaultfd into the page fault handling of container processes. First, the checkpoint restore system CRIU [23] already provides a migration functionality that uses userfaultfd. With lazy migration, a container can be migrated to another host without the transfer of memory pages. When the container begins execution and requests a memory page on the previous host, a userfaultfd page fault event will be triggered and the page can be fetched over the network. CRIU lazy migration could be integrated with FluidMem, such that the destination host is the same as the source. The checkpoint action would store all pages in a FluidMem key-value store, and when the container starts, the FluidMem page fault handler would intercept page faults and fetch memory from the remote key-value store. This combination of CRIU and FluidMem would allow containers to be disaggregated in addition to VMs. There would be no need to evict operating system pages, since they are not part of the container. However, because memory operations such as copy-on-write, fork(), and madivse() would be exposed directly to userfaultfd, CRIU developers have expressed concern about handling race conditions [24].

A second approach would be through a new userfaultfd device driver written as a kernel module. When a page fault occurs in a container, the kernel expects the backing memory to reside on a hardware DIMM module. There is no difference between a container page fault and that of a native process. A new feature in Linux 4.18 called Heterogeneous Memory Management (HMM) [40] allows address spaces between a device and the kernel to be mirrored. If a container could be instructed use memory from HMM via CRIU or memory hotplug, that address space could be mirrored in a device that traps all those page faults to userfaultfd. Work has been done to integrate HMM with memory hotplug, but for right now, kernel developers are opposed to allowing persistent memory to be accessed like native DRAM.

Chapter 7

Conclusion

This thesis has articulated the need for performant, complete, and flexible memory disaggregation in virtualized cloud computing environments and detailed a novel system architecture capable of realizing these needs in current data centers. The FluidMem implementation and its optimizations were presented with a detailed investigation of page fault latency. Evaluations on applications Graph500 and MongoDB demonstrate that FluidMem outperforms the swap-based alternatives used by existing memory disaggregation research.

7.1 Contributions

The contributions of this thesis are improvements in three areas of memory disaggregation with evaluations relevant to the context of virtualized cloud computing environments:

- **Performance**: the FluidMem user space page fault handler with RAMCloud improves performance over paging via the swap interface to an RDMA block device.
- **Completeness**: full memory disaggregation is made possible with FluidMem's ability to incrementally size up and down a VM's footprint.
- **Flexibility**: an open source page fault handler in user space integrates with RAMCloud and memcached key-value stores and allows for easy customization by cloud operators.

7.2 Priorities for future work

Towards expanding FluidMem's contributions and usefulness in current and future cloud environments, two priorities emerge for future work.

7.2.1 Sizing VM's by working set size

The novel capability of full memory disaggregation in the cloud may bring experimental methodology to the cloud operator's art of balancing the oversubscription of hypervisor resources. As discussed in Section 6.8, FluidMem would allow for the working set size to be estimated even when it is below the VM size. The impact of lowering performance by decreasing the VM footprint would have to be balanced with benefits gained from knowing the working set size. However, with FluidMem's easy downsizing of VMs, this problem could be explored programmatically.

7.2.2 Memory disaggregation with containers

Since container platforms such as Docker [25] are becoming more common in the data center, the strategies suggested in Section 6.9 could be a path towards memory disaggregation for high-performance computing and cluster environments that are not virtualized. Many of the difficulties lie with low-level interfaces of the Linux kernel, but the relevant areas of persistent memory and HMM are changing rapidly and may be influenced by memory disaggregation implementations.

7.3 Concluding remarks

Despite the concept of expanding local memory to include memory having been extensively researched in the past, this thesis has identified the set of challenges applicable to cloud data centers and presents a novel architecture along with its implementation, FluidMem, to provide performant, complete, and flexible memory disaggregation in the cloud. As a fully open source project integrated with an open source cloud virtualization stack, it is the author's hope that FluidMem will be readily adopted in cloud data centers and enable new use cases.

Bibliography

[1] Adve, S. V., and Gharachorloo, K. Shared Memory Consistency Models: A Tutorial. Computer 29, 12 (Dec. 1996), 66–76.

[2] Agarwal, A., Chaiken, D., Johnson, K., Kranz, D., Kubiatowicz, J., Kurihara, K., Lim, B. H., Maa, G., Nussbaum, D., Parkin, M., and Yeung, D. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1991.

[3] Aguilera, M. K., Amit, N., Calciu, I., Deguillard, X., Gandhi, J., Subrahmanyam, P., Suresh, L., Tati, K., Venkatasubramanian, R., and Wei, M. Remote Memory in the Age of Fast Networks. In Proceedings of the 2017 Symposium on Cloud Computing (New York, NY, USA, 2017), SoCC '17, ACM, pp. 121–127.

[4] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12) (San Jose, CA, 2012), USENIX, pp. 253–266.

[5] Amit, N., Tsafrir, D., and Schuster, A. VSwapper: A Memory Swapper for Virtualized Environments. In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 349–366.

[6] Microsoft Azure. `http://azure.microsoft.com/`.

[7] Bennett, J. K., Carter, J. B., and Zwaenepoel, W. Munin: Distributed shared memory based on type-specific memory coherence. In Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (New York, NY, USA, 1990), PPOPP '90, ACM, pp. 168–176.

[8] Bielski, M., Syrigos, I., Katrinis, K., Syrivelis, D., Reale, A., Theodoropoulos, D., Alachiotis, N., Pnevmatikatos, D., Pap, E. H., Zervas, G., Mishra, V., Saljoghei, A., Rigo, A., Zazo, J. F., Lopez-Buedo, S., Torrents, M., Zyulkyarov, F., Enrico, M., and de Dios, O. G. dReDBox: Materializing a full-stack rack-scale system prototype of a next-generation disaggregated datacenter. In 2018 Design, Automation Test in Europe Conference Exhibition (DATE) (March 2018), pp. 1093–1098.

[9] Black, D. L., Milojičić, D. S., Dean, R. W., Dominijanni, M., Langerman, A., and Sears, S. J. Extended Memory Management (XMM): Lessons Learned. Softw. Pract. Exper. 28, 9 (July 1998), 1011–1031.

[10] Bobroff, N., Kochut, A., and Beaty, K. Dynamic Placement of Virtual Machines for Managing SLA Violations. In 2007 10th IFIP/IEEE International Symposium on Integrated Network Management (May 2007), pp. 119–128.

[11] Buluç, A., and Madduri, K. Parallel Breadth-first Search on Distributed Memory Systems. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (New York, NY, USA, 2011), SC '11, ACM, pp. 65:1–65:12.

[12] Chae, D., Kim, J., Kim, Y., Kim, J., Chang, K. A., Suh, S. B., and Lee, H. CloudSwap: A cloud-assisted swap mechanism for mobile devices. In 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) (May 2016), pp. 462–472.

[13] Chamberlain, B., Callahan, D., and Zima, H. Parallel Programmability and the Chapel Language. Int. J. High Perform. Comput. Appl. 21, 3 (Aug. 2007), 291–312.

[14] Chapman, M., and Heiser, G. vNUMA: A Virtual Shared-memory Multiprocessor. In Proceedings of the 2009 Conference on USENIX Annual Technical Conference (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 2–2.

[15] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. X10: An object-oriented approach to non-uniform cluster computing. SIGPLAN Not. 40, 10 (Oct. 2005), 519–538.

[16] Chen, H., Luo, Y., Wang, X., Zhang, B., Sun, Y., and Wang, Z. A transparent remote paging model for virtual machines. In International Workshop on Virtualization Technology (2008).

[17] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live Migration of Virtual Machines. In Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (Berkeley, CA, USA, 2005), NSDI'05, USENIX Association, pp. 273–286.

[18] Comer, D. E., and Griffioen, J. A new design for distributed systems: The remote memory model. In Proceedings of the Summer 1990 USENIX Conference (June 1990).

[19] Compeau, P. E. C., Pevzner, P. A., and Tesler, G. How to apply de Bruijn graphs to genome assembly. Nat Biotech 29, 11 (Nov 2011), 987–991.

[20] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[21] Corbalan, J., Martorell, X., and Labarta, J. Evaluation of the memory page migration influence in the system performance: the case of the SGI O2000. In Proceedings of the 17th annual international conference on Supercomputing (2003), ACM, pp. 121–129.

[22] Costa, P., Ballani, H., and Narayanan, D. Rethinking the Network Stack for Rack-scale Computers. In Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing (Berkeley, CA, USA, 2014), HotCloud'14, USENIX Association, pp. 12–12.

[23] CRIU. https://criu.org.

[24] CRIU: userfaultfd. https://criu.org/Userfaultfd.

[25] Docker. https://www.docker.com/.

[26] DPDK. http://dpdk.org.

[27] Dragojevi, A., Narayanan, D., Hodson, O., and Castro, M. FaRM: Fast remote memory. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI (2014), vol. 14.

[28] Ekman, M., and Stenstrom, P. A Cost-Effective Main Memory Organization for Future Servers. In 19th IEEE International Parallel and Distributed Processing Symposium (April 2005), pp. 45a–45a.

[29] Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., and Lehner, W. SAP HANA Database: Data Management for Modern Business Applications. SIGMOD Rec. 40, 4 (Jan. 2012), 45–51.

[30] Feeley, M. J., Morgan, W. E., Pighin, E. P., Karlin, A. R., Levy, H. M., and Thekkath, C. A. Implementing Global Memory Management in a Workstation Cluster. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (New York, NY, USA, 1995), SOSP '95, ACM, pp. 201–212.

[31] Fleisch, B., and Popek, G. Mirage: A Coherent Distributed Shared Memory Design. SIGOPS Oper. Syst. Rev. 23, 5 (Nov. 1989), 211–223.

[32] Fleisch, B., and Popek, G. Mirage: A coherent distributed shared memory design. In Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (New York, NY, USA, 1989), SOSP '89, ACM, pp. 211–223.

[33] Fleisch, B. D., Hyde, R. L., and Juul, N. C. Mirage+: A Kernel Implementation of Distributed Shared Memory on a Network of Personal Computers. Softw. Pract. Exper. 24, 10 (Oct. 1994), 887–909.

[34] Flouris, M. D., and Markatos, E. P. The Network RamDisk: Using Remote Memory on Heterogeneous NOWs. Cluster Computing 2, 4 (Oct. 1999), 281–293.

[35] FluidMem: open source full memory disaggregation. https://github.com/blakecaldwell/fluidmem.

[36] Forin, A., Forin, R., Barrera, J., Young, M., and Rashid, R. Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach. Tech. rep., In 1988 Winter USENIX Conference, 1988.

[37] Gao, P. X., Narayan, A., Karandikar, S., Carreira, J., Han, S., Agarwal, R., Ratnasamy, S., and Shenker, S. Network Requirements for Resource Disaggregation. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) (GA, 2016), USENIX Association, pp. 249–264.

[38] Gu, J., Lee, Y., Zhang, Y., Chowdhury, M., and Shin, K. G. Efficient Memory Disaggregation with Infiniswap. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17) (Boston, MA, 2017), USENIX Association, pp. 649–667.

[39] Han, S., Egi, N., Panda, A., Ratnasamy, S., Shi, G., and Shenker, S. Network Support for Resource Disaggregation in Next-generation Datacenters. In Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks (New York, NY, USA, 2013), HotNets-XII, ACM, pp. 10:1–10:7.

[40] Heterogeneous Memory Management. `https://www.kernel.org/doc/html/v4.18/vm/hmm.html`.

[41] Hines, M. R., and Gopalan, K. MemX: Supporting large memory workloads in Xen virtual machines. In Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing (New York, NY, USA, 2007), VTDC '07, ACM, pp. 2:1–2:8.

[42] Hwang, J., Uppal, A., Wood, T., and Huang, H. Mortar: Filling the Gaps in Data Center Memory. SIGPLAN Not. 49, 7 (Mar. 2014), 53–64.

[43] Hyper-V. `https://www.microsoft.com/en-us/cloud-platform/server-virtualization`.

[44] Kablan, M., Caldwell, B., Han, R., Jamjoom, H., and Keller, E. Stateless Network Functions. In Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (New York, NY, USA, 2015), HotMiddlebox '15, ACM, pp. 49–54.

[45] Kachelmeier, L. A., Van Wig, F. V., and Erickson, K. N. Comparison of High Performance Network Options: EDR InfiniBand vs.100Gb RDMA Capable Ethernet. Tech. rep., Los Alamos National Laboratory, 8 2016.

[46] Keleher, P., Cox, A. L., Dwarkadas, S., and Zwaenepoel, W. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (Berkeley, CA, USA, 1994), WTEC'94, USENIX Association, pp. 10–10.

[47] Kerrighed Webpage. `http://kerrighed.org`.

[48] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click Modular Router. ACM Trans. Comput. Syst. 18, 3 (Aug. 2000), 263–297.

[49] Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., and Hennessy, J. The Stanford FLASH Multiprocessor. In Proceedings of the 21st Annual International Symposium on Computer Architecture (Los Alamitos, CA, USA, 1994), ISCA '94, IEEE Computer Society Press, pp. 302–313.

[50] KVM Hypervisor. http://www.linux-kvm.org/.

[51] Laudon, J., and Lenoski, D. The SGI Origin: A ccNUMA highly scalable server. In Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on (June 1997), pp. 241–251.

[52] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. S. The Stanford Dash Multiprocessor. Computer 25, 3 (Mar. 1992), 63–79.

[53] LevelDB. http://github.com/google/leveldb.

[54] Li, K., and Hudak, P. Memory Coherence in Shared Virtual Memory Systems. ACM Trans. Comput. Syst. 7, 4 (Nov. 1989), 321–359.

[55] Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., Li, S., Yang, H., Wang, J., and Wang, J. De novo assembly of human genomes with massively parallel short read sequencing. Genome Res 20, 2 (Feb 2010), 265–272. 20019144[pmid].

[56] Liang, S., Noronha, R., and Panda, D. Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device. In Cluster Computing, 2005. IEEE International (Sept. 2005), pp. 1 –10.

[57] libvirt: virtualization API. https://libvirt.org/.

[58] Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S. K., and Wenisch, T. F. Disaggregated Memory for Expansion and Sharing in Blade Servers. SIGARCH Comput. Archit. News 37, 3 (June 2009), 267–278.

[59] Lim, K., Turner, Y., Santos, J. R., AuYoung, A., Chang, J., Ranganathan, P., and Wenisch, T. F. System-level Implications of Disaggregated Memory. In Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (Washington, DC, USA, 2012), HPCA '12, IEEE Computer Society, pp. 1–12.

[60] LMDB. https://symas.com/lmdb/.

[61] Luo, R., Liu, B., Xie, Y., Li, Z., Huang, W., Yuan, J., He, G., Chen, Y., Pan, Q., Liu, Y., Tang, J., Wu, G., Zhang, H., Shi, Y., Liu, Y., Yu, C., Wang, B., Lu, Y., Han, C., Cheung, D. W., Yiu, S.-M., Peng, S., Xiaoqian, Z., Liu, G., Liao, X., Li, Y., Yang, H., Wang, J., Lam, T.-W., and Wang, J. SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. GigaScience 1, 1 (2012), 18.

[62] Ma, Z., Sheng, Z., and Gu, L. DVM: A Big Virtual Machine for Cloud Computing. IEEE Transactions on Computers 63, 9 (Sept 2014), 2245–2258.

[63] Markatos, E. P., and Dramitinos, G. Implementation of a reliable remote memory pager. In Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (Berkeley, CA, USA, 1996), ATEC '96, USENIX Association, pp. 15–15.

[64] McSherry, F., Isard, M., and Murray, D. G. Scalability! But at What Cost? In Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (Berkeley, CA, USA, 2015), HOTOS'15, USENIX Association, pp. 14–14.

[65] Memcached. http://memcached.org.

[66] Midorikawa, H., Kurokawa, M., Himeno, R., and Sato, M. DLM: A distributed large memory system using remote memory swapping over cluster nodes. In 2008 IEEE International Conference on Cluster Computing (Sept 2008), pp. 268–273.

[67] Mitchell, C., Geng, Y., and Li, J. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13) (San Jose, CA, 2013), USENIX, pp. 103–114.

[68] monetdb. https://www.monetdb.org/.

[69] MongoDB. https://www.mongodb.com/.

[70] MongoDB: Production Notes. https://docs.mongodb.com/manual/administration/production-notes/.

[71] Morin, C., Gallard, P., Lottiaux, R., and Vallée, G. Towards an efficient single system image cluster operating system. Future Generation Computer Systems 20, 4 (2004), 505–521.

[72] Mosix Webpage. http://mosix.org.

[73] Murphy, R. C., Wheeler, K. B., Barrett, B. W., and Ang, J. A. Introducing the Graph 500. In Cray Users Group (CUG) (2010).

[74] Accelio based network block device. https://github.com/accelio/NBDX/.

[75] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S., and Oskin, M. Latency-tolerant Software Distributed Shared Memory. In Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (Berkeley, CA, USA, 2015), USENIX ATC '15, USENIX Association, pp. 291–305.

[76] Newhall, T., Finney, S., Ganchev, K., and Spiegel, M. Nswap: A Network Swapping Module for Linux Clusters. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 1160–1169.

[77] Nitu, V., Kocharyan, A., Yaya, H., Tchana, A., Hagimont, D., and Astsatryan, H. Working Set Size Estimation Techniques in Virtualized Environments: One Size Does Not Fit All. Proc. ACM Meas. Anal. Comput. Syst. 2, 1 (Apr. 2018), 19:1–19:22.

[78] Nitu, V., Teabe, B., Tchana, A., Isci, C., and Hagimont, D. Welcome to Zombieland: Practical and Energy-efficient Memory Disaggregation in a Datacenter. In Proceedings of the Thirteenth EuroSys Conference (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 16:1–16:12.

[79] NVM Express over Fabrics 1.0a. http://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1_0a-2018.07.23-Ratified.pdf.

[80] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., and Rosenblum, M. Fast Crash Recovery in RAMCloud. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (New York, NY, USA, 2011), SOSP '11, ACM, pp. 29–41.

[81] OpenSSI Webpage. http://openssi.org.

[82] OpenStack. https://www.openstack.org/.

[83] Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[84] Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., Rumble, S., Stutsman, R., and Yang, S. The RAMCloud Storage System. ACM Trans. Comput. Syst. 33, 3 (Aug. 2015), 7:1–7:55.

[85] Papagiannis, A., Saloustros, G., González-Férez, P., and Bilas, A. An Efficient Memory-Mapped Key-Value Store for Flash Storage. In Proceedings of the ACM Symposium on Cloud Computing (New York, NY, USA, 2018), SoCC '18, ACM, pp. 490–502.

[86] Persistent Memory Programming. https://pmem.io/.

[87] Qemu: the FAST! processor emulator. https://www.qemu.org/.

[88] RackSpace Openstack Public Cloud. https://www.rackspace.com/en-us/openstack/public.

[89] RAMACHANDRAN, U., and KHALIDI, M. Y. A. An Implementation of Distributed Shared Memory. SOFTWAREPRACTICE AND EXPERIENCE 21, 5 (1991), 443–464.

[90] Rao, P. S., and Porter, G. Is Memory Disaggregation Feasible?: A Case Study with Spark SQL. In Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems (New York, NY, USA, 2016), ANCS '16, ACM, pp. 75–80.

[91] Redis. http://redis.io/.

[92] Rizzo, L. Netmap: A Novel Framework for Fast Packet I/O. In Proceedings of the 2012 USENIX Conference on Annual Technical Conference (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 9–9.

[93] Salah, K., Al-Shaikh, R., and Sindi, M. Towards Green Computing Using Diskless High Performance Clusters. In Proceedings of the 7th International Conference on Network and Services Management (Laxenburg, Austria, Austria, 2011), CNSM '11, International Federation for Information Processing, pp. 456–459.

[94] Salzberg, S. L., Phillippy, A. M., Zimin, A., Puiu, D., Magoc, T., Koren, S., Treangen, T. J., Schatz, M. C., Delcher, A. L., Roberts, M., Marçais, G., Pop, M., and Yorke, J. A. GAGE: A critical evaluation of genome assemblies and assembly algorithms. Genome Res 22, 3 (Mar 2012), 557–567. 22147368[pmid].

[95] Samih, A., Wang, R., Maciocco, C., Tai, T.-Y. C., Duan, R., Duan, J., and Solihin, Y. Evaluating Dynamics and Bottlenecks of Memory Collaboration in Cluster Systems. In Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012) (Washington, DC, USA, 2012), CCGRID '12, IEEE Computer Society, pp. 107–114.

[96] ScaleMP. http://www.scalemp.com/.

[97] Schatz, M. C., Delcher, A. L., and Salzberg, S. L. Assembly of large genomes using second-generation sequencing. Genome Res 20, 9 (Sep 2010), 1165–1173. 20508146[pmid].

[98] Scott, S. L. Synchronization and Communication in the T3E Multiprocessor. In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA, 1996), ASPLOS VII, ACM, pp. 26–36.

[99] SGI UV 3000, UV 30. https://www.sgi.com/pdfs/4555.pdf.

[100] Shan, Y., Huang, Y., Chen, Y., and Zhang, Y. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18) (Carlsbad, CA, 2018), USENIX Association, pp. 69–87.

[101] Sites, R. L. An Analysis of the Cray-1 Computer. In Proceedings of the 5th Annual Symposium on Computer Architecture (New York, NY, USA, 1978), ISCA '78, ACM, pp. 101–106.

[102] Smith, R., and Rixner, S. A Policy-based System for Dynamic Scaling of Virtual Machine Memory Reservations. In Proceedings of the 2017 Symposium on Cloud Computing (New York, NY, USA, 2017), SoCC '17, ACM, pp. 282–294.

[103] Souto, P., and Stark, E. W. A Distributed Shared Memory Facility for FreeBSD. In Proceedings of the Annual Conference on USENIX Annual Technical Conference (Berkeley, CA, USA, 1997), ATEC '97, USENIX Association, pp. 11–11.

[104] SQLite: Memory-Mapped I/O. https://www.sqlite.org/mmap.html.

[105] Suetake, M., Kashiwagi, T., Kizu, H., and Kourai, K. S-memV: Split Migration of Large-Memory Virtual Machines in IaaS Clouds. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD) (Jul 2018), vol. 00, pp. 285–293.

[106] Making swapping scalable. https://lwn.net/Articles/704478/.

[107] TidalScale. https://www.tidalscale.com/.

[108] UFFDIO REMAP kernel patches. https://github.com/blakecaldwell/userfault-kernel/tree/userfault_4.20-rc7.

[109] Userfaultfd Kernel Documentation. http://www.kernel.org/doc/Documentation/vm/userfaultfd.txt.

[110] VoltDB. http://www.voltdb.com/.

[111] Waldspurger, C. A. Memory Resource Management in VMware ESX Server. SIGOPS Oper. Syst. Rev. 36, SI (Dec. 2002), 181–194.

[112] Wang, Z., Wang, X., Hou, F., Luo, Y., and Wang, Z. Dynamic Memory Balancing for Virtualization. ACM Trans. Archit. Code Optim. 13, 1 (Mar. 2016), 2:1–2:25.

[113] Wood, T., Shenoy, P. J., Venkataramani, A., Yousif, M. S., et al. Black-box and Gray-box Strategies for Virtual Machine Migration. In NSDI (2007), vol. 7, pp. 17–17.

[114] Xen Project. http://www.xenproject.org/.

[115] Xu, J., Cui, L., Hao, Z., and Wang, C. SA-PFRS: Semantics-Aware Page Frame Reclamation System in Virtualized Environments. In 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS) (Dec 2017), pp. 684–693.

[116] Yang, J., and Seymour, J. Pmbench: A Micro-Benchmark for Profiling Paging Performance on a System with Low-Latency SSDs. In Information Technology-New Generations. Springer, 2018, pp. 627–633.

[117] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.

[118] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12) (San Jose, CA, 2012), USENIX, pp. 15–28.

[119] Zerbino, D. R., and Birney, E. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Res. 18, 5 (May 2008), 821–829.

[120] Zhang, D., Ehsan, M., Ferdman, M., and Sion, R. DIMMer: A Case for Turning off DIMMs in Clouds. In Proceedings of the ACM Symposium on Cloud Computing (New York, NY, USA, 2014), SOCC '14, ACM, pp. 11:1–11:8.

[121] Zhang, Q., Liu, L., Su, G., and Iyengar, A. MemFlex: A Shared Memory Swapper for High Performance VM Execution. IEEE Transactions on Computers 66, 9 (Sep. 2017), 1645–1652.

[122] ZONE DEVICE and the future of struct page. https://lwn.net/Articles/717555/.