FluidMem: Full, Flexible, and Fast Memory Disaggregation for the Cloud

Blake Caldwell*, Sepideh Goodarzy,
Sangtae Ha, Richard Han, Eric Keller, Eric Rozner
Department of Computer Science
University of Colorado Boulder
Boulder, Colorado, USA
first.last@colorado.edu

Youngbin Im
School of Electrical and Computer Engineering
UNIST
Ulsan, South Korea
ybim@unist.ac.kr

***Abstract*—This paper presents a new approach to memory disaggregation called FluidMem that leverages the user-fault mechanism in Linux to achieve full memory disaggregation in software. FluidMem enables dynamic and transparent resizing of an unmodified Virtual Machine's (VM's) memory footprint in the cloud. As a result, a VM's memory footprint can seamlessly scale over multiple machines or even be downsized to a near-zero footprint on a given server. FluidMem's architecture provides flexibility to cloud operators to manage remote memory without requiring guest intervention, while also supporting paging out the entirety of a VM's pages within its address space. FluidMem integrates with a remote memory backend in a modular way, easily supporting systems such as RAMCloud to harness remote memory. We demonstrate FluidMem outperforms an existing memory disaggregation approach based on network swap. Microbenchmarks are evaluated to characterize the latency of different components of the FluidMem architecture, and two memory-intensive applications are demonstrated using FluidMem, the Graph500 benchmark, and MongoDB. Additionally, we show FluidMem can flexibly and efficiently grow and shrink the memory footprint of a VM as defined by a cloud provider.**

***Keywords*-memory disaggregation; virtualization; cloud**

I. Introduction

While fast random access memory repeatedly increases in speed and capacity, the memory demands of applications also continue to match or outpace availability. Whether for genomic [1], [2], enterprise [3], storage [4], or data analytic purposes [5], memory-intensive applications continually require large memory footprints to satisfy performance requirements. Without sufficient memory, performance suffers, or in the worst case, out-of-memory (OOM) errors are encountered. Developers and administrators typically have little recourse when encountering insufficient memory: options range from reinitiating workloads, to manually adjusting configurations, to modifying code, and to eventually giving up and paying for more memory capacity. None of these solutions are ideal, and therefore decades of research have long envisioned leveraging underutilized memory in a cluster of servers [6], [7].

Early work to share memory suffered from poor performance (e.g., Distributed Shared Memory [8], [9], [10]), required significant re-writes of operating systems (e.g., single system images [11], [12]), or required applications to be modified to explicitly deal with remote memory (e.g., with key-value stores [13], [14], [15]). Recent work on *memory disaggregation*, however, takes a fresh look at the problem. In the disaggregation model, computational units can be composed of discrete quantities of memory from many different servers. Hardware designs are being investigated to provide this model natively [16], [17], [18], [19], but require new datacenter infrastructure. The case for a software approach has received renewed attention [20], [21] due to advances in network technology. Some systems such as Infiniswap [22] are transparent to applications and work with existing operating systems by leveraging the Linux swap interface. Swap suffers from a key limitation, however, because it only provides *partial memory disaggregation.*

This paper introduces FluidMem, a system that provides **full memory disaggregation in software** for VM-based workloads that requires no changes in hardware design. There are two key differences between full and partial memory disaggregation. First, with full memory disaggregation, *all* memory pages are capable of being disaggregated (able to be stored in any server). Remote paging with swap only enables anonymous pages to be disaggregated, and therefore many other pages such as file-backed pages (e.g., allocated with `mmap`) or unevictable pages (e.g., pinned memory, kernel memory) cannot be disaggregated. FluidMem can support full memory disaggregation for any of a VM's pages. Second, full memory disaggregation can decouple memory management from the entity using the memory, the VM, with no requirements of guest assistance. As a result, a cloud provider can manage the memory of tenant VMs running an unmodified operating system (OS), without needing to coordinate with the guest OS or processes running in the VM. Together, full memory disaggregation enables a provider to both increase *and* decrease memory allocation, enabling a VM's memory footprint to scale to many machines or be downsized to nearly zero memory on a given server.

We have implemented FluidMem by leveraging a new Linux kernel feature userfaultfd [23]. Intuitively, supporting software-based full memory disaggregation would come at

* At Brown University since completion of this work

a cost when compared to software-based partial memory disaggregation approaches. In reality, this is not the case when comparing FluidMem with swap-based approaches. FluidMem's performance compares favorably to swap-based approaches because full memory disaggregation moves unused operating system pages out of DRAM and FluidMem efficiently rearranges the ordering of operations in a page fault, reducing the latency of the page fault handling critical path. Our microbenchmarks show that page fault latencies via FluidMem to RAMCloud are 40% faster than the NVMe over Fabrics [24] remote memory swap device and 77% faster than SSD swap. Our macro benchmarks demonstrate that FluidMem outperforms swap-based remote memory used by existing memory disaggregation implementations in a MongoDB [4] workload and the Graph500 [25] benchmark. Finally, we demonstrate the scale of downsizing possible with full disaggregation. With the memory footprint reduced to 180 pages (720 KB), a VM can still respond and open up an SSH shell.

In summary, this paper makes the following contributions:

- We introduce the notion of software-based full memory disaggregation, allowing any page of a VM's memory footprint to be stored anywhere in the datacenter.
- We present a new design to realize full memory disaggregation transparently, requiring no changes to the VM, while allowing providers to scale or restrict a VM memory footprint across machines.
- We implement FluidMem and compare to swap-based schemes, showing favorable performance while enabling more flexible memory management. Fluidmem's code is available on GitHub [26].

The outline of this paper is as follows. Section II first motivates full memory disaggregation. Section III provides an overview of FluidMem, while Sections IV-V provide architectural details. FluidMem is evaluated in Section VI, and related work is covered in Section VII before concluding.

II. Motivating Software-based Full Memory Disaggregation

Resource disaggregation should be transparent to the software running on the disaggregated system. This transparency model matches the deployment model in cloud computing, which decouples the provider running the infrastructure from the tenant running its software. With disaggregation, a cloud provider can assemble computing resources to an exact tenant specification, and then provision complete control of software using those resources to the tenant. Providers can allocate assembled resources as a one-time build, or even dynamically change the assembly by adding more memory, CPU, and storage. Given the promise of such flexibility, industry and academia alike are researching new hardware architectures that can efficiently support this model of full disaggregation [16], [17], [18], [19].

Our goal is to provide full memory disaggregation *in software*. That is, rather than re-architecting system hardware to build systems on-demand from disparate resources, we instead utilize a software layer to provide the same abstraction with resources pooled across a collection of commodity servers. Today, virtualization software runs on a single physical server to provide the abstraction of a machine built on-demand for a given tenant. With full memory disaggregation, the virtualization layer extends usage beyond a single physical server to include memory from multiple machines. This approach enables cloud providers to allocate memory to a tenant's VM and then dynamically manage the assignment of the VM's memory in a deterministic fashion to physical resources throughout the cloud infrastructure.

To motivate the need for a new approach to memory disaggregation, we step back and discuss the leading alternative to achieve transparent memory disaggregation – remote paging with swap [22], [27]. The swap mechanism moves pages between main memory and disk, and its design provides a convenient translation layer between memory and block sectors. By assigning a block device to reside over a network, swap-based memory disaggregation enables remote memory use without modifying VMs. The key limitation, however, is that it is difficult to support full memory disaggregation with swap. First, under low memory conditions, file-backed memory pages are written to the original filesystem. The same functionality occurs for pages in memory-mapped regions created by the `mmap` system call, commonly used to store file executables in memory and by some in-memory databases [28]. There is no capability to store these pages in swap space. Further, swap limits the pages that can be successfully swapped out, even if they are unused. Pages belonging to the kernel are one such category of non-swappable pages. There are also unevictable pages, such as pages pinned by the `mlock` system call. With swap-based disaggregation approaches, these pages are unable to utilize remote memory, and thus swap-based memory disaggregation approaches cannot provide full memory disaggregation.

Additionally, swap-based approaches are unable to completely decouple memory management from the VM using the memory without explicit VM support. For example, there is no way to reduce a VM's local memory footprint on a server at any given time. Such functionality can be realized with VM ballooning (see Section VII), but this requires VM modification and cooperation. Without ballooning, swap-based memory disaggregation doesn't activate until there is high memory pressure on a machine, minimizing opportunities to proactively disaggregate memory. Typically swap replaces pages based on LRU information, meaning it is agnostic to a VM's current footprint and cannot easily implement a provider's or application's custom memory usage policy. This limits the ability of a provider to flexibly manage memory allocations.

Next, we motivate why FluidMem supports full memory disaggregation for VMs. With the advent of lightweight VMs [29], VMs can be put to sleep often (and resumed on a whim), scaled to support microservice models, or migrated quickly. Today, providers like Amazon provision lightweight VMs to house transient workloads like serverless functions with Firecracker [30]. On the opposite end of the spectrum, VMs may still represent their large, monolithic ancestors, with a specific VM requiring a large memory footprint, or supporting an application that aims to process large amounts of data in memory. FluidMem seamlessly enables full memory disaggregation for all of these deployment scenarios, even in cases were hypervisor swap is disabled, as in Firecracker [31] deployments.

Finally, one could imagine an approach that seeks full memory disaggregation using existing (swap-based) approaches. In particular, if a VM is run inside a container, then an approach like Infiniswap could manage the memory of the container (and thus the VM) through swap. One downside is other functionalities that support the VM (e.g., QEMU) will be loaded within the container and will be grouped within the container's memory. This is both dangerous, as critical memory supporting virtualization can be swapped out, and wasteful, as extra memory beyond what is used by the application becomes disaggregated. One might consider this suitable, but we argue that a design to realize full memory disaggregation in software should match the model as cleanly as possible.

III. FluidMem Architecture

We aim to create a software layer that enables full memory disaggregation on existing computer systems. Rather than co-opting the swap interface, we find a more efficient path in Linux memory management that naturally allows full memory disaggregation and granular management of hypervisor memory. In this way, memory pages can natively be stored in a key-value store on a remote server.

Three key questions need to be answered to collectively define how FluidMem works. We highlight these below and then describe the detailed structure and mechanics of FluidMem in subsequent sections. An overview of the FluidMem system architecture is illustrated in Figure 1.

How can FluidMem manage memory without explicit VM support? Transparency to tenant VMs is an important goal in memory disaggregation for the cloud. While VMs (including the operating system) are typically defined by the user, the hypervisor emulator (e.g. QEMU) and hypervisor kernel (e.g. Linux/KVM) are managed by the cloud operator. Since minimal changes to the hypervisor stack will be tolerated by the cloud provider for reliability and security reasons, we have chosen to define an interface of registering memory regions with FluidMem that is analogous to registering local memory from the perspective of the hypervisor.

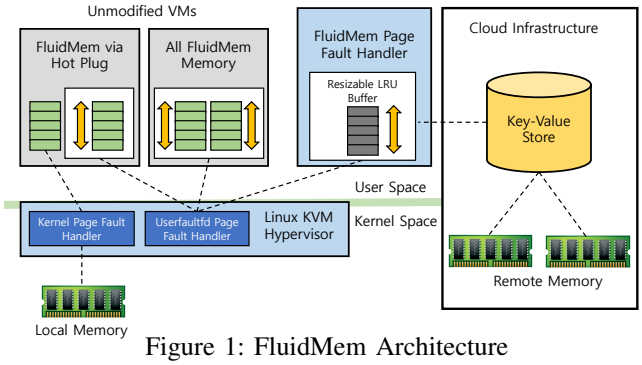


Figure 1: FluidMem Architecture

FluidMem can manage memory requested by the hypervisor in two basic modes: a normal VM can add extra FluidMem memory via *memory hotplug* (left VM in Figure 1) or a VM can be started with all its memory registered with the FluidMem page fault handler (right VM in Figure 1). VMs backed with FluidMem memory are completely unmodified. Hotplug is natively supported in Linux, Windows, and FreeBSD guest VMs through QEMU [32], which means memory can be added to a VM at any time, even if the VM did not anticipate using additional memory at boot time.

The guest kernel views FluidMem memory as if it were standard physical memory. VM memory that is not registered with FluidMem is serviced by the standard kernel page fault handler and uses DRAM memory local to the hypervisor. In FluidMem-registered regions, memory accesses will pass through the FluidMem page fault handler and be routed to the appropriate location in a remote key-value store. Full memory disaggregation is enabled when all VM memory has been registered with VM.

How can FluidMem achieve transparent page fault handling? As with swap, FluidMem implements a page fault mechanism that allows memory accesses to be fast (when accesses are in local memory) but also supports the use of remote memory in a key-value store. In contrast to swap, FluidMem leverages the *userfaultfd* feature in the Linux kernel [23], supported since Linux 4.3 and originally designed to support VM live migration. This allows us to directly tap into the kernel's page fault handling mechanism, and because of this, we can disaggregate all memory pages, unlike swap. Further, as we directly handle the page faults in user space, there are a variety of optimizations that are immediately possible. In particular, no additional context switch is needed for user space network transport protocols like RAMCloud. Additionally, libraries such as Boost and the Zookeeper client can easily be linked in to build indexing structures or synchronize cluster state.

How can FluidMem dynamically manage the allocation of local and remote memory resources? A key motivation of full memory disaggregation is that it allows the cloud provider to dynamically manage the memory of a tenant VM

transparently. FluidMem uses the userfaultfd mechanism to track all FluidMem-registered memory. Unlike swap, userfaultfd is invoked on the first page fault of every page, giving the user space page fault handler the ability to identify all pages belonging to a VM. An administrator can then manage VM memory allocations in a fine-grained manner, dynamically mapping VM memory between local and remote memory pages. The userfaultfd capability allows the local memory buffer to be actively sized up or down to balance the demands of the VM's workload with the resource constraints or policies of the cloud operator.

Cloud providers can further benefit from the flexibility that comes from handling memory paging in user space to rapidly deploy a variety of customizations needed for their infrastructure or specific use cases. Some examples are page compression or replication across remote servers.

IV. Expanding To Remote Memory

In this section, we describe how FluidMem registers remote-backed memory and supports storing memory pages in a key-value store.

FluidMem's scheme for expanding to remote memory is implemented in QEMU by wrapping the allocation of a guest VM's memory with an allocation that also registers the memory region with the FluidMem user space page fault handler. Registration is accomplished via the `userfaultfd` system call, which returns a file descriptor that is monitored for page fault events. The size of the memory allocation is the amount of physical memory that appears in the guest VM. This wrapper function is provided in a user functions library component of FluidMem that is dynamically linked to QEMU. Other hypervisors besides QEMU could become FluidMem-enabled by linking the same library.

FluidMem interfaces with key-value stores via a generic API that supports partitions and allows multiple VMs to share the same key-value store. For networked key-value stores such as RAMCloud [33] and Memcached [34] that natively support partitions, we use their user space clients for preparing PUT and GET requests. The 4 KB page contents serve as the value portion of the request and the key is a 64-bit integer matching the first 52 bits of the virtual memory address used by the faulting application (e.g. QEMU). This is adequate to uniquely represent each 4 KB page in the 64-bit virtual address space. To support other key-value stores without partition support, we use the remaining 12 bits to index a "virtual partition". The index is created using the process PID, a hypervisor ID, and a nonce, where global uniqueness is ensured by a replicated and globally consistent table stored in *Zookeeper* [35].

V. Fast Handling of Remote Page Faults

This section describes in greater detail the path followed by a page fault through the FluidMem page fault handler and optimizations made to reduce page fault latency.

A. User space page fault handler

The process that is responsible for handling page faults in FluidMem runs entirely in user space and is called the *monitor process*. Its primary responsibility is to watch for page faults and resolve them before waking up the faulting process. The monitor process waits on a list of file descriptors (corresponding to registered userfaultfd regions) for events indicating a userfaultfd page fault. The list of descriptors is extended whenever a new region is registered (VM started) and shrunk when regions become invalid (VM shut down). The memory region initially contains no mapped pages so any access to an address within the range will trigger a page fault. Below in Figure 2, we show a trace of the components involved in handling a first-time access.

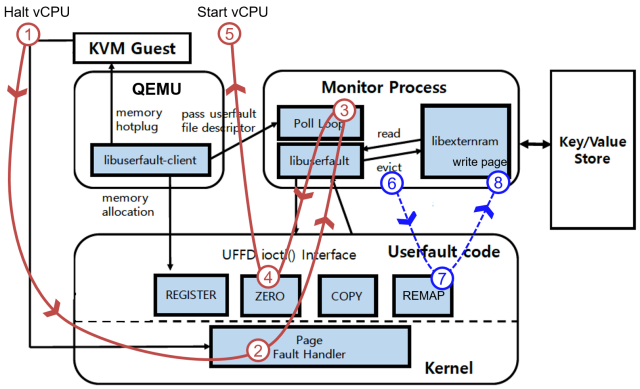


Figure 2: First page access handling critical path (red) begins when the guest is halted (1) as a page fault occurs (2). The monitor process is notified of the fault via a userfaultfd event (3). With FluidMem's *pagetracker* feature, the monitor process only has to make a `UFFD_ZERO ioctl` for the zero-filled page (4) before waking up the QEMU guest (5). Asynchronous (blue) page eviction (6) is accomplished by moving the page out of the VM via `UFFD_REMAP` (7) and writing the page to a key-value store (8).

When a QEMU/KVM virtual machine accesses an address without a corresponding page mapping (i.e. when the VM is booting), the vCPU thread within the process must be halted until the fault can be resolved. This appears as a page fault to the VM at the guest virtual address, but the hypervisor will see the page fault at a virtual address belonging to the VM's QEMU process and will run userfaultfd-specific handling code before sending an event to the monitor process via a file descriptor. On notification of a page fault, the monitor receives the faulting address and the process PID belonging to the VM. The monitor keeps a list of already seen pages to avoid reads from the remote key-value store for first-time accesses. Instead, the fault is resolved by placing the special

zero-filled page[1] at the faulting address and then resuming execution of the vCPU thread.

The monitor maintains an LRU list to manage page evictions, where the size of the list determines the number of pages held in DRAM for all VMs. Evictions come from the top of the LRU list and will be triggered by the monitor process when the number of pages reaches the configured maximum size and another page fault arrives. Note that the LRU list is only updated when a page is seen by the monitor process, which only happens on first access and after an eviction. A future optimization would be to trigger faults for pages not yet evicted to the key-value store. At present, the internal ordering of the list does not change.

An advantage of using the built-in userfaultfd kernel feature is that pages in userfaultfd regions can be transparently managed like other kernel pages. For example, in emergency situations, it would be possible for VM pages to be paged to swap space on the hypervisor without intervention by the monitor process. We assume that the remote key-value store determines if its pages should remain resident in DRAM. RAMCloud, for example, pins memory to ensure that it is not paged out. On the other hand, it may be desirable to allow remote memory to spill over to disk, NVRAM, or another storage medium. To perform evictions, we use a proposed `UFFD_REMAP` operation with userfaultfd to remove a page from the VM and place it in a user space buffer by changing page table entries[2]. The monitor process can then send the page to the key-value store.

When a page is accessed for the second time after having been evicted to a key-value store, a slightly different path is taken. This time the monitor process notes that the faulting page address has been seen before, so it issues a read to the key-value store. After a successful read, the page is copied into the VM and the vCPU woken up. Our early technical report on FluidMem contains more details on how the monitor process handles page faults [36].

B. Optimizations to page fault handling

This section describes various performance optimizations to FluidMem's page fault handling.

Asynchronous writeback: Asynchronous writeback to the key-value store is an important optimization made to FluidMem because evicted pages do not need to be written to the key-value store immediately as long as they are not requested by the guest VM. Rather than waiting for the write to complete before handling the next page fault, the critical path in the monitor only evicts the page from the VM and puts the page on a write list before moving on to the next fault. A separate thread periodically flushes the write list to the key-value store when its size has reached a configured batch size of pages or a stale file descriptor has been found. We leverage RAMCloud's multi-write operation to write batches of pages belonging to the same userfaultfd region. This optimization is most beneficial when slower network transports are used for the key-value store such as with TCP with Memcached.

In a related optimization, the page fault handler can steal pages from the pending write list to resolve a page fault and shortcut two round trips to the remote key-value store. If a write of a page is in-flight when the fault handler gets another fault for the same address, there is no other choice than to wait for the write to complete. However, the critical path will resume immediately once the pending write has completed.

Asynchronous reads: Other than when a VM first boots up, reads are most often accompanied by an eviction to maintain a constant memory footprint. Even though eviction with `UFFD_REMAP` moves a page from inside the VM to outside by only modifying page table entries and not copying the page contents, it needs to synchronize processor page tables for KVM guests using interprocessor interrupts. In our microbenchmarks, we found this call took 4-5 μs. Combined with the observation that a page read from RAMCloud involved waiting (10 μs) for the network transport, we saw the opportunity to interleave the eviction operation with the network read. Both RAMCloud and Memcached read operations were split into top and bottom halves, making use of their respective asynchronous API calls. This optimization reduced overall CPU usage by running `UFFD_REMAP` at a time when the vCPU thread was already suspended, and the `UFFD_REMAP` call returned after only 2 μs.

The asynchronous optimizations above are similar to the optimizations already present in the kernel swap interface where kernel threads decouple eviction from the read critical path.

Zero-copy semantics: A benefit that stems from operating entirely in one context (user space or kernel space) is that data copy operations can be avoided. The `UFFD_REMAP` operation demonstrates this, but it is not always faster than `UFFD_COPY` because of the synchronization required. We took care throughout the page fault handling code to avoid copies and reuse buffers. Such copying is necessary when using the swap interface because a memory page must be put into a block device request and traverse several layers of kernel code before reaching a custom kernel module to use remote memory.

This benefit has not fully been realized in our implementation because RAMCloud does not use RDMA network transport and incurs a copy into the Infiniband network buffer. Substituting RAMCloud for an RDMA key-value store such as FaRM [13] or HERD [37] would further reduce FluidMem's page fault latency.

[1]The zero page within the kernel is a copy-on-write page that returns all zeroes on read, but on a subsequent write fault, it will trigger a regular page fault to allocate a normal empty page allocation.

[2]Our patches have been submitted to the Linux kernel mailing list for upstream inclusion and work is ongoing to merge them into mainline Linux.

VI. Evaluation

This section examines the performance of FluidMem. We first describe the experimental platform and comparison points made to current memory disaggregation strategies. Next, we run microbenchmarks of page fault latency within a VM, then investigate code contributions to latency and performance improvements from our optimizations. Lastly, we demonstrate FluidMem's performance with the Graph500 benchmark [25] and the document store MongoDB [4].

A. Test platform configuration

These experiments were conducted 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 FDR Infiniband (56 Gb/s) network connected the nodes with Mellanox ConnectX-3 cards. The FluidMem with RAMCloud and NVMe over Fabrics (NVMeoF) [24] tests used native Infiniband transport and the FluidMem with Memcached [34] tests were run with the IP over IB transport.

RAMCloud is given 25 GB of memory and is running on a different server than the one running the test VM. The replication feature with RAMCloud was not turned on, but replication only impacts key-value writes. Since FluidMem carries out writes asynchronously, the overall impact on page fault latency would be minimal.

To understand FluidMem's performance relative to systems like Infiniswap [22], we used a swap device backed by remote DRAM on another machine with NVMe over Fabrics (NVMeoF). Both NVMeoF and Infiniswap enable swap-based memory disaggregation using RDMA, where the NVMeoF project has gained acceptance into the Linux kernel as a successor to the NBDx block device [38]. Since Infiniswap requires building kernel modules for each new kernel release and has code specific to kernel versions, it is not clear how it will be used on cloud servers that continually need to be updated with security patches. The Infiniswap block device did not have kernel modules that would work with Linux 4.20, so we used swap backed by local DRAM in our evaluations as a lower bound for swap-based approaches.

NVMeoF's predecessor, NBDx, is compared to Infiniswap's RDMA network block device in [22] with bandwidth results favorable to Infiniswap. The reason given for this difference was high remote CPU usage, which is reduced with Infiniswap. We measured CPU usage of the remote NVMeoF system while running pmbench [39] and observed that peak kernel CPU usage for NVMeoF never exceeded 20%. Thus, we do not expect that our latency benchmarks were limited by high CPU usage. Nonetheless, Infiniswap to a remote machine would be significantly slower than swap backed by local DRAM with no network latency.

The NVMeoF block device is made available on the hypervisor by loading a kernel module that connects to the remote target via Infiniband. The NVMeoF storage target was a different server accessible via FDR Infiniband. The target device size was 10 GB of DRAM via `/dev/pmem0` [40].

B. Latency micro-benchmarks

To better understand the raw page fault latency seen by different FluidMem backends and how they influenced application performance in Section VI-D, we measured access latencies using pmbench [39] within a VM. Our goal using pmbench was to measure access microsecond latencies with minimal overhead and to examine their distributions, which are shown in Figure 3. The working set size (WSS) was set by a 4 GB allocation from pmbench. First, pmbench warms up the cache by accessing all pages once, and then randomly makes 4 KB requests at a 50% read to write ratio for 100 s.

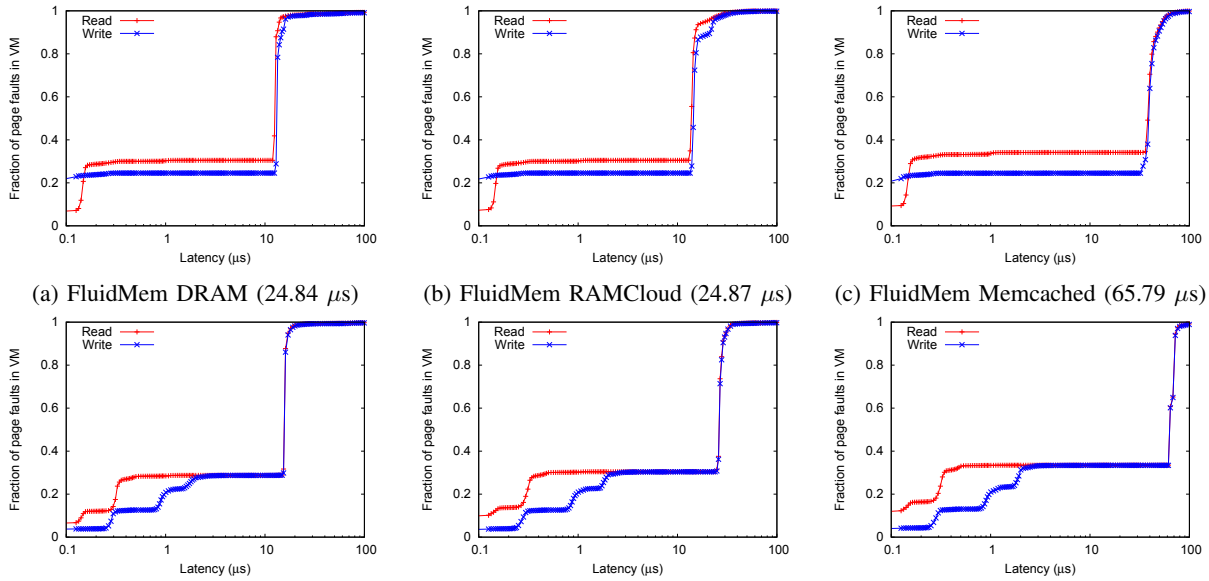
For the swap cases, a 20 GB block device backed by either DRAM, NVMeoF, or SSD was made accessible to the VM as swap space in addition to 1 GB of DRAM. When FluidMem was used, we registered the VM's memory with the FluidMem page fault handler on boot, meaning all memory accesses were served by FluidMem. The monitor process enforced a 1 GB LRU list size, 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 5 GB, but maintaining 1 GB in DRAM.

Figure 3 shows each backend's cumulative distribution function (CDF) of page fault latencies. The average latency is lower with DRAM-backed FluidMem compared to DRAM-backed swap. For remote memory configurations, latency is reduced by 40% with FluidMem over NVMeoF with swap. The similarity between DRAM and RAMCloud backend performance with FluidMem indicates that the optimizations were effective for hiding network latency.

The CDFs highlight differences in the page fault handling paths used by FluidMem and swap. Any page fault that took less than 10 μs must have been cached in DRAM (slightly over 25% from the local to remote memory ratio). The early part of the CDF reflects latencies from page faults that do not require a network round trip to resolve. For FluidMem, these accesses are represented in the flat section of the CDF that ends before 10.5 μs. In the swap case, there are multiple flat parts corresponding to a more complex page fault path. In all cases, the rise to 100% starts between 10 and 100 μs and is a reflection of the performance characteristics of the remote memory backend. We see similar remote memory performance between FluidMem and swap scenarios.

C. FluidMem optimizations

FluidMem has the built-in ability to profile individual components of the page fault handling path. We used this to profile key sections of FluidMem code during synchronous page fault handling (without the optimizations in Table II). Results from this analysis are shown in Table I with the RAMCloud backend. A takeaway is that reducing the time waiting for network read and write operations holds the most



(a) FluidMem DRAM (24.84 μs) (b) FluidMem RAMCloud (24.87 μs) (c) FluidMem Memcached (65.79 μs)

(d) Swap DRAM (26.34 μs) (e) Swap NVMeoF (41.73 μs) (f) Swap SSD (106.56 μs)

Figure 3: CDF of latencies measured with pmbench. The plots are arranged row-wise by mechanism (FluidMem vs. Swap). Average latencies for each backend are shown in parenthesis.

potential for decreasing overall latency. FluidMem's cache management functions make relatively small contributions to total latency as compared to network operations. Note that the 99th percentile latency for `UFFD_REMAP` is high because the operation requires an interrupt to be sent to all CPUs to flush the TLB entry.

To examine application performance implications, we measured the time between entry and exit points in the kernel's page fault handler when faults are generated by a simple test program that reads from and writes to a memory region. Memory can be accessed sequentially or randomly. The program was linked with FluidMem's *libuserfault* library, so there was no involvement of a virtualization layer. We used the Linux `perf` command to measure the time the kernel took to resolve each page fault with various optimization enabled. Average latencies with RAMCloud are shown in Table II. The improvements come from enabling the monitor process to hide the network latency by interleaving asynchronous operations in the page fault handling path. An improvement from baseline to fully-optimized with DRAM indicates that interleaving the userfaultfd system calls was helpful even without a network latency component. Comparing latencies between DRAM and RAMCloud show that a network key-value store incurs a 20-40% overhead.

D. Application Use Cases

This subsection describes use cases of standard applications running on FluidMem and demonstrates the performance improvement from expanding a VM's physical memory in contrast to only increasing swap space.

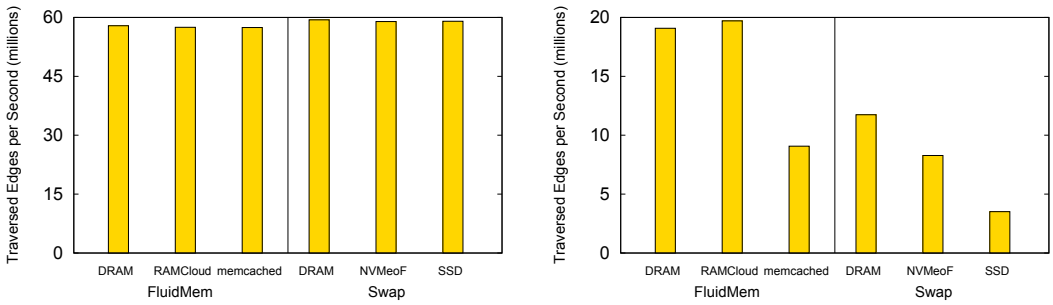
Table I: Latencies of key parts of FluidMem code involved when the page is accessed (units: μs).

Code path	Latency		
	Avg	Stdev	99th
`UPDATE_PAGE_CACHE`	2.56	0.25	3.32
`INSERT_PAGE_HASH_NODE`	2.58	1.26	8.36
`INSERT_LRU_CACHE_NODE`	2.87	0.47	3.65
`UFFD_ZEROPAGE`	2.61	0.44	3.51
`UFFD_REMAP`	1.65	2.57	18.03
`UFFD_COPY`	3.89	0.77	5.43
`READ_PAGE`	15.62	31.01	20.90
`WRITE_PAGE`	14.70	1.52	17.45

Table II: Average page fault latencies measured from the application with various FluidMem optimizations (units: μs).

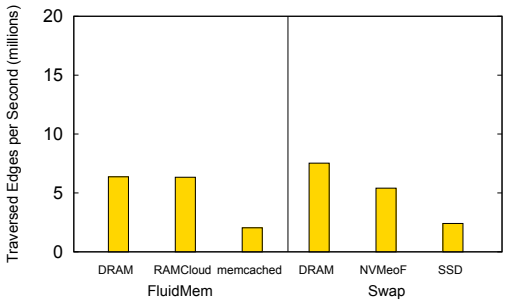
	FluidMem with DRAM		FluidMem with RAMCloud	
Optimization	Sequential	Random	Sequential	Random
Default	27.25	28.15	66.71	58.70
Async Read	25.26	25.00	51.08	49.33
Async Write	23.67	30.26	42.88	43.40
Async Read/Write	21.30	24.37	29.47	29.20

Applications that load large datasets or indexes into memory will benefit greatly from FluidMem when the application's WSS is free to grow beyond local DRAM and expand to remote memory. While some applications can easily partition their working set into discrete chunks and spread them across the aggregate DRAM of multiple nodes, the heterogeneity of cloud applications means that this assumption cannot always be made. If the capability to partition data does not exist without explicit refactoring of the application's code, or if a partitioning method is not known beforehand, FluidMem can still improve performance by

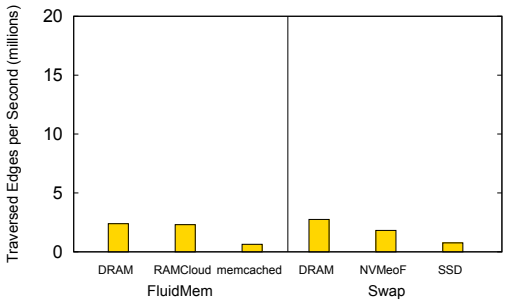


(a) WSS 60% of DRAM (scale factor 20)

(b) WSS 120% of DRAM (scale factor 21)



(c) WSS 240% of DRAM (scale factor 22)



(d) WSS 480% of DRAM (scale factor 23)

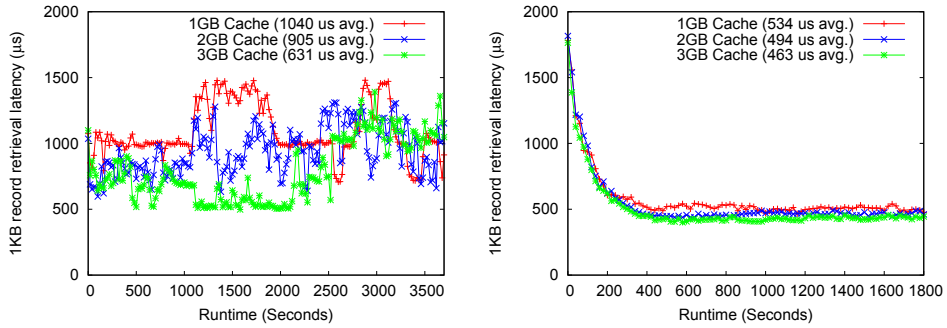
Figure 4: Graph500 performance with working set sizes (WSS) from 600 MB (scale factor 20) to 4.8 GB (scale factor 23). The overhead of FluidMem page faults is 2.6% in (a). The benefit of storing OS pages in remote memory with FluidMem is most pronounced in (b) when WSS is 120% of DRAM. At higher scale factors (c) and (d), the performance of FluidMem RAMCloud exceeds swap to NVMeoF, but choices in page eviction penalize FluidMem DRAM compared to swap.

allowing more of the dataset to reside in DRAM. FluidMem provides an alternative solution to loading these datasets into memory that doesn't require engineer development effort.

1) Graph500: We chose the Graph500 benchmark to evaluate how page fault latency on various key-value stores 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 [41]. For this reason, we used the sequential reference implementation of Graph500 [25]. We note that there are many prior works on parallel BFS on distributed clusters [41], but the focus of this paper is on full memory disaggregation in the cloud, limited to individual VMs, not a distributed shared address space.

All experiments were run on a QEMU/KVM virtual machine with 2 vCPUs and 1 GB of local memory on the hypervisor. For swap, this meant a memory capacity of 1 GB for the VM. Block devices backed by different mediums 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 was not involved. This setting was critical for an accurate comparison between swap and FluidMem. With the disk caching mode set to "writeback", writes to the swap device would be buffered in the hypervisor's cache. Using "writeback" actually made swapping to DRAM slower because of the extra caching layer. For FluidMem, 1 GB of DRAM was used by fixing the LRU list size before adding 4 GB of remote memory via hotplug. Swap was turned off for FluidMem tests.

Figure 4 shows the results of the Graph500 benchmark run on VMs configured to use FluidMem or swap, each with three different backends. The benchmark creates a graph in memory of configurable size and then performs 64 consecutive BFS traversals. The scale factor controls the size of the graph which in our evaluation ranges from 1 GB (scale factor 20) to 5 GB (scale factor 20). Performance is measured using the metric (millions of) traversed edges per second (TEPS). For each configuration, the harmonic mean of TEPS for the 64 trials is reported in Figure 4.



(a) Read latency with swap (NVMeoF) (b) Read latency with FluidMem (RAMCloud)

Figure 5: Latency of YCSB 1 KB read-only workload for MongoDB with WiredTiger storage engine. When cache size exceeds DRAM capacity, the storage engine is not capable of achieving a stable working set when only given swap space via NVMeoF. When remote memory performance is comparable (Figure 3), FluidMem achieves significantly lower average latencies because it transparently provides the storage engine with native memory capacity.

The purpose of varying the scale factor was to evaluate performance when the application WSS fits entirely within DRAM (600 MB for scale factor 20) to when the WSS necessitates storing a majority of frequently used pages remotely (scale factor 23 uses 4.8 GB). The results in Figure 4 can generalize to a larger VM with a higher scale factor by comparing the percentage of WSS that can remain in DRAM. Note that the memory footprint of the OS is approximately 300 MB of DRAM at boot (shown in Table III), which would become a smaller percentage memory overhead with a larger VM.

Figure 4a shows the harmonic mean TEPS at scale factor 20. Since this test involved purely local accesses, it was used to assess the overhead of FluidMem's full memory disaggregation. Since FluidMem traps to user space and performs a hash lookup for each page it hasn't seen before, the cost of the kernel triggering a "minor page fault" with FluidMem is slightly higher than with swap. At scale factor 20, the number of minor page faults was about 150,000, but this only resulted in a 2.6% slowdown with FluidMem.

The FluidMem configurations do significantly better than swap-based ns at scale factor 21 where the WSS occupies 120% of local memory (Figure 4b). The large performance difference is primarily because FluidMem allows more unused kernel pages to be removed from DRAM and replaced with useful application pages. Another aspect of Figure 4b that has promise for cloud datacenters with standard Ethernet networks is that the Memcached backend for FluidMem performs better than swap backed by NVMeoF 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 with SSD (Figure 3).

At scale factor 22, FluidMem with RAMCloud outperforms swap with NVMeoF, consistent with lower remote memory latencies measured in Figure 3. However, the DRAM backed storage via swap is slightly faster than DRAM through FluidMem. Since we have shown above that page fault latencies for FluidMem backed by DRAM are lower than swap backed by DRAM, we believe the difference is a result of the `kswapd` process within the guest being better able to pick candidates for eviction using the kernel's active/inactive list mechanism. This is a current limitation of our LRU list design mentioned in section V-A.

The same relative comparisons hold at scale factor 23 (WSS 480% of local memory). While the ability to choose pages to evict may give swap an edge when network latency is low (e.g. DRAM), the FluidMem page fault handler effectively hides network latency with RAMCloud compared to swap with NVMeoF.

Beyond scale factor 23, where the WSS takes up more than 480% of DRAM, Graph500 will still run to completion. However, other applications could impose timeouts on certain operations that will be exceeded when using remote memory. Infiniswap 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. Applications such as Spark likely have timeouts that cause such failures and changing the application code to account for the remote memory delay could resolve them. In addition to the 480% WSS case, we explored an analogous situation, but at the opposite extreme, where a VM is booted with a DRAM footprint of fewer than 200 pages (1 MB). This is discussed further in Section VI-E

2) MongoDB: Not all applications can take advantage of extra memory through the swap interface which is necessary to benefit from remote memory through systems such as Infiniswap. Full memory disaggregation through FluidMem allows applications such as MongoDB to efficiently use remote memory even if applications have their own cache management system that is incompatible with swap.

	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 III: Summary of the effects of reducing VM footprint to 1 page

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 *WiredTiger* that uses an application-specific cache and the kernel's filesystem cache. We only evaluate WiredTiger here.

One of the limitations of swap is that it cannot be used to store memory-mapped pages. When an application uses `mmap` with remote memory via swap, 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 via swap. While MongoDB has deprecated the *mmapv1* storage engine, emerging data stores continue to use `mmap` [28]. We point out that FluidMem can benefit such applications that continue to use `mmap`.

We chose the read-only workload from the Yahoo Cloud Serving Benchmark (YCSB) suite [42], for our evaluation of MongoDB. With this workload, data will be cached in memory until evicted to make room for newly read records (1 KB each). For swap, the MongoDB server is run on a VM with 1 GB of local DRAM. The swap device is backed by either DRAM on a remote server via `/dev/pmem0`, an NVMeoF target device, or a local SSD partition. For FluidMem, the VM was created with 4 GB of memory, but it was limited to a DRAM footprint of 1 GB by the LRU list. Swap space was turned off for tests with FluidMem. For all test configurations, transparent huge pages and NUMA were disabled. Since, for swap configurations, we used remote memory rather than a local hard drive, `vm.swappiness` and disk readahead were set to 100 and 0, respectively.

YCSB workload C (read-only) was run from the same VM to reduce the impact of network latency on the results. Since 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 was used, where the kernel was free to schedule the MongoDB and YCSB processes on any vCPU. Before initiating the measured experiment, dataset records were inserted into the MongoDB store by YCSB. Once loading was complete, the storage engine used approximately 5 GB on a local SSD. To ensure that the WiredTiger memory cache and kernel page cache had been flushed, the VM was rebooted between tests.

The time courses of read latency for the WiredTiger storage engine on swap backed by NVMeoF and FluidMem backed by RAMCloud are shown in Figures 5a and 5b, respectively. The latency measurements represent the time it takes to read a 1 KB record from MongoDB. Some of the records will require disk I/O, while others can be read from the in-memory cache. 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 establishing a stable working set in memory. The poor interaction between the WiredTiger storage engine's memory cache and `kswapd` leads to 36-95% higher average latencies than with FluidMem.

E. Full memory disaggregation

Table III demonstrates FluidMem's ability to perform full memory disaggregation by minimizing a VM's memory footprint. Virtual machines may remain on, but unused, and cloud providers could benefit from a mechanism to repurpose idle memory capacity for increasing density. Users may opt for a billing model that allows them to keep VMs reachable, with the capacity to scale up on demand.

Without FluidMem enforcing the LRU list size, a VM will consume 317 MB of memory just from booting to a command prompt. This memory will be resident in DRAM and the usage will grow over time up to the allotted memory size of the VM. An alternative for reducing the memory footprint is KVM's balloon driver for reclaiming guest pages, but the driver reaches its maximum size when the VM footprint is still 64 MB. In contrast, FluidMem's LRU list can be resized to enforce a near-zero footprint. When the footprint is reduced to 180 pages (0.7 MB), the VM is still able to accept SSH logins before a timeout. Even part of the ssh binary will have to be stored in FluidMem, along with all libraries and kernel code needed to complete a user authentication. Afterward, if the LRU size is increased, the VM will instantly return to normal responsiveness.

At only 80 pages, the VM can still respond to an ICMP echo request every 1 s. Within the 1 s interval, the VM is able to retrieve the network packet and send out a response. Below 80 pages, ICMP requests will queue up, but will still trigger a response if the footprint is increased again.

To reduce the footprint down to 1 page, full virtualization using QEMU [32] was used to keep the VM functional, though it appeared non-responsive. Increasing the footprint would make the VM usable again. We suspect there was a deadlock in the page fault handling with KVM hardware-assisted virtualization since handling a page fault can trigger more page faults. With full virtualization, the recursive triggering of page faults would still succeed.

VII. Related Work

Distributed Shared Memory (DSM) Some of the earliest implementations of shared memory were found in DSM hardware-based systems. Due to the high cost of such systems, many software-based DSM systems were proposed [8], [9], [10]. They provide a single address space, like hardware DSM systems, but added protocols for sharing and invalidation into the OS. Network and coordination overheads posed a problem for the design and performance of these protocols. 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 PGAS systems, allow for the existence of multiple, separate key domains [43], [44]. PGAS approaches, however, require modifying application code, and hence have limited applicability to cloud settings.

Single System Images (SSI) SSI implementations provide the abstraction of a machine comprised of resources from many nodes [11]. They require heavyweight OS modifications and implement DSM to provide a single address space with coherency guarantees. Linux SSI implementations [45] run individual processes, so they don't provide the transparency benefits of VMs as used in cloud computing today. They are also incompatible with current Linux kernels.

Remote-backed swap space The transparent use of remote memory by unmodified applications can be realized using a custom kernel module for a block device backed by memory. The device is then configured as the swap device [22], [27]. The design space is broad, with works supporting VMs [27], containers [22], and utilizing a variety of technologies like Ethernet [27] and RDMA networks [22]. As described earlier, swap-based approaches to memory disaggregation cannot provide full memory disaggregation like FluidMem.

System resource disaggregation Rack-scale memory disaggregation approaches include hardware prototypes [17], a Xen hypervisor-based implementation of remote paging [46], and a new kernel [47] that implements a memory transfer service for system resource disaggregation. All of these require major changes to existing cloud hardware or software stacks. Like FluidMem, dReDBox [17] uses memory hotplug to add additional memory, but does not expose a mechanism to provide decreasing memory capacity for full memory disaggregation. Remote swap devices that implement memory disaggregation [21], [22] provide a transparent way to add memory, but are not able to provide native memory to benefit applications like MongoDB, nor do they provide full memory disaggregation.

VM Ballooning A technique to dynamically balance memory usage among virtual machines is using a balloon driver [48]. This kernel module installed in the guest VM coordinates with the hypervisor to either inflate or deflate its memory allocations within the guest. Ballooning takes a relatively long time to reclaim pages used within the guest because they must be flushed to disk before they can be reused. Other work has improved performance with intelligent prefetching, cache replacement, and fair share algorithms [49]. Ballooning approaches, however, require explicit VM cooperation or modifying applications in the VM, whereas FluidMem works with unmodified VMs.

Hypervisor paging Hypervisor paging allows the hypervisor to directly swap out guest physical memory, guaranteeing a set amount of memory to be reclaimed on short timescales. As opposed to ballooning, hypervisor paging is done without full information of pages used by the guest so the wrong choices for reclaiming can be expensive. VSwapper [50] modifies the hypervisor by adding the visibility necessary to avoid common swap inefficiencies. Other work shows swapping pages back in is expensive and moves pages into a shared memory swap device on the host to reduce overheads [51]. None of these techniques examine full memory disaggregation like FluidMem. Other research [52] modified the hypervisor to provide disaggregated memory. The downsides with their method are the requirement of a custom hypervisor and that their solution is in kernel space.

Live migration (LM) LM [53] can relocate an entire VM, including its memory footprint, to another hypervisor. LM and memory disaggregation are complementary since LM is capable of moving execution and memory disaggregation can offload memory from the hypervisor. The emulation software QEMU provides a LM technique called post-copy migration that uses *userfaultfd* [23]. In this way, a VM can be migrated before all its memory pages have been copied. When the VM starts on the remote host and accesses a page left behind, the page fault is caught by *userfaultfd* and the missing page is retrieved over the network. Unlike QEMU, FluidMem was designed for full memory disaggregation.

VIII. Conclusion

This paper has presented and motivated a novel solution for software-based full memory disaggregation in virtualized cloud computing environments. Our technique, called FluidMem, achieves full memory disaggregation by enabling any page of a VM to be moved to remote memory. Importantly, full memory disaggregation is enabled without any support of the guest VM. Combined, these properties ensure any page within a VM can be stored or serviced anywhere within

the datacenter. Our scheme allows the memory footprint of a VM to seamlessly expand across multiple machines and even enables a provider to shrink the memory footprint of a given VM to near zero on a server. As opposed to swap-based memory disaggregation techniques, FluidMem makes novel use of the *userfaultfd* page fault handler to efficiently and flexibly enable full memory disaggregation. 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.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation (NSF) under CNS grant 1337399 and CAREER award 1652698. Support was also provided through the 2019 Research Fund (1.190149.01) of UNIST. Finally, we would like the thank William Mortl, Kannan Subramanian, and Daniel Zurawski for their FluidMem code and testing contributions.

REFERENCES

[1] R. Luo *et al.*, "SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler," *GigaScience*, vol. 1, no. 1, p. 18, 2012.

[2] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de Bruijn graphs," *Genome Res.*, vol. 18, no. 5, pp. 821–829, May 2008.

[3] F. Färber *et al.*, "SAP HANA Database: Data Management for Modern Business Applications," *SIGMOD Rec.*, vol. 40, no. 4, pp. 45–51, Jan. 2012.

[4] "MongoDB," https://www.mongodb.com/.

[5] M. Zaharia *et al.*, "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: USENIX, 2012, pp. 15–28.

[6] D. E. Comer and J. Griffioen, "A new design for distributed systems: The remote memory model," in *Proceedings of the Summer 1990 USENIX Conference*, June 1990.

[7] A. Samih *et al.*, "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: IEEE Computer Society, 2012, pp. 107–114.

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

[9] B. D. Fleisch, R. L. Hyde, and N. C. Juul, "Mirage+: A Kernel Implementation of Distributed Shared Memory on a Network of Personal Computers," *Softw. Pract. Exper.*, vol. 24, no. 10, pp. 887–909, Oct. 1994. [Online]. Available: http://dx.doi.org/10.1002/spe.4380241003

[10] P. Souto and E. W. Stark, "A Distributed Shared Memory Facility for FreeBSD," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '97. Berkeley, CA, USA: USENIX Association, 1997, p. 11.

[11] C. Morin, P. Gallard, R. Lottiaux, and G. Vallée, "Towards an efficient single system image cluster operating system," *Future Generation Computer Systems*, vol. 20, no. 4, pp. 505–521, 2004.

[12] Z. Ma, Z. Sheng, and L. Gu, "DVM: A Big Virtual Machine for Cloud Computing," *IEEE Transactions on Computers*, vol. 63, no. 9, pp. 2245–2258, Sept 2014.

[13] A. Dragojević *et al.*, "FaRM: Fast remote memory," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI*, vol. 14, 2014.

[14] J. Ousterhout *et al.*, "The RAMCloud Storage System," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015.

[15] C. Mitchell, Y. Geng, and J. Li, "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: USENIX, 2013, pp. 103–114.

[16] P. Costa, H. Ballani, and D. Narayanan, "Rethinking the Network Stack for Rack-scale Computers," in *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'14. Berkeley, CA, USA: USENIX Association, 2014, p. 12.

[17] M. Bielski *et al.*, "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.

[18] H. Volos *et al.*, "Memory-oriented distributed computing at rack scale," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: ACM, 2018, p. 529.

[19] K. Lim *et al.*, "Disaggregated Memory for Expansion and Sharing in Blade Servers," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 267–278, Jun. 2009.

[20] M. K. Aguilera *et al.*, "Remote Memory in the Age of Fast Networks," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 121–127. [Online]. Available: http://doi.acm.org/10.1145/3127479.3131612

[21] P. X. Gao *et al.*, "Network Requirements for Resource Disaggregation," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 249–264.

[22] J. Gu *et al.*, "Efficient Memory Disaggregation with Infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 649–667.

[23] "Userfaultfd Kernel Documentation," https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html.

[24] "NVM Express over Fabrics," https://nvmexpress.org/.

[25] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," in *Cray User's Group*, 2010.

[26] "FluidMem," https://github.com/blakecaldwell/fluidmem.

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

[28] A. Papagiannis *et al.*, "An Efficient Memory-Mapped Key-Value Store for Flash Storage," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: ACM, 2018, pp. 490–502.

[29] F. Manco *et al.*, "My VM is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 218–233.

[30] "Firecracker," https://firecracker-microvm.github.io.

[31] "Firecracker Production Host Setup Recommendations," https://github.com/firecracker-microvm/firecracker/blob/master/docs/prod-host-setup.md.

[32] "QEMU," http://www.qemu.org.

[33] D. Ongaro *et al.*, "Fast Crash Recovery in RAMCloud," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 29–41.

[34] "Memcached," http://memcached.org.

[35] P. Hunt *et al.*, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, p. 11.

[36] B. Caldwell, Y. Im, S. Ha, R. Han, and E. Keller, "Fluidmem: Memory as a service for the datacenter," *CoRR*, vol. abs/1707.07780, 2017. [Online]. Available: http://arxiv.org/abs/1707.07780

[37] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 295–306, Aug. 2014.

[38] "Accelio based network block device," https://github.com/accelio/NBDX/.

[39] J. Yang and J. Seymour, "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.

[40] "Persistent Memory Programming," https://pmem.io/.

[41] A. Buluç and K. Madduri, "Parallel Breadth-first Search on Distributed Memory Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, p. 65.

[42] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[43] P. Charles *et al.*, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005.

[44] J. Nelson *et al.*, "Latency-tolerant Software Distributed Shared Memory," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '15. Berkeley, CA, USA: USENIX Association, 2015, pp. 291–305.

[45] "Kerrighed," http://www.kerrighed.org.

[46] K. Lim *et al.*, "System-level Implications of Disaggregated Memory," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, ser. HPCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2012.6168955

[47] Y. Shan *et al.*, "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 69–87.

[48] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844146

[49] J. Hwang *et al.*, "Mortar: Filling the Gaps in Data Center Memory," *SIGPLAN Not.*, vol. 49, no. 7, pp. 53–64, Mar. 2014. [Online]. Available: http://doi.acm.org/10.1145/2674025.2576203

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

[51] Q. Zhang *et al.*, "MemFlex: A Shared Memory Swapper for High Performance VM Execution," *IEEE Transactions on Computers*, vol. 66, no. 9, pp. 1645–1652, Sep. 2017.

[52] K. Koh *et al.*, "Disaggregated cloud memory with elastic block management," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 39–52, 2019.

[53] C. Clark *et al.*, "Live Migration of Virtual Machines," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286.