Can Far Memory Improve Job Throughput?

Emmanuel Amaro
UC Berkeley

Christopher Branner-Augmon
UC Berkeley

Zhihong Luo
UC Berkeley

Amy Ousterhout
UC Berkeley

Marcos K. Aguilera
VMware Research

Aurojit Panda
New York University

Sylvia Ratnasamy
UC Berkeley

Scott Shenker
UC Berkeley & ICSI

Abstract

As memory requirements grow, and advances in memory technology slow, the availability of sufficient main memory is increasingly the bottleneck in large compute clusters. One solution to this is memory disaggregation, where jobs can remotely access memory on other servers, or *far memory*. This paper first presents faster swapping mechanisms and a far memory-aware cluster scheduler that make it possible to support far memory at rack scale. Then, it examines the conditions under which this use of far memory can increase job throughput. We find that while far memory is not a panacea, for memory-intensive workloads it can provide performance improvements on the order of 10% or more even without changing the total amount of memory available.

ACM Reference Format:

Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. **Can Far Memory Improve Job Throughput?. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3342195.3387522**

1 Introduction

The rising popularity of in-memory workloads such as machine learning applications and key-value stores is causing memory demands in compute clusters to grow rapidly [14]. At the same time, because of the end of Moore's law, DRAM manufacturers are struggling to achieve higher storage densities and lower per-storage-unit costs [40, 44]. Taken together, these two developments cause main memory to increasingly be the bottleneck when operating compute clusters [24, 43].

Memory disaggregation, which has been the subject of both academic research [2, 30, 34, 41, 48, 49, 58] and commercial projects [6, 19, 20], is one way to address the memory bottleneck, as it allows compute nodes to access memory at remote nodes; we will call this *far memory*. While far memory does not reduce the total amount of memory needed to run individual jobs, nor does it make memory cheaper or more dense, far memory does mean that jobs need not be restricted to *local* memory but instead can utilize memory that is located elsewhere in the cluster. This works around the "memory capacity wall" [48] and increases the extent to which memory can be shared efficiently across jobs.

While there has been previous work investigating how far memory can impact individual jobs [34], there has been limited work on how effective far memory is in actually increasing job throughput, or equivalently, in reducing job makespans for a cluster workload: that is the issue we address in this paper. There are many ways one can support far memory, including making the local memory in each server available for remote use. However, for specificity and simplicity, we consider one particular model of memory disaggregation: using one or more "memory servers" to support all far memory, while all other servers in the cluster use their memory to support local jobs.

There are two main barriers to making far memory practical. The first barrier lies in how one designs the swapping mechanisms needed to access far memory, as existing systems [30, 34, 47] that swap over RDMA suffer from poor latency and throughput due to head-of-line blocking, and to handling interrupts and page reclamation on the critical path of page fault resolution. We have designed a Linux swap system, called **Fastswap**, that is optimized to use far memory through RDMA. Like other swap systems, it is transparent both to applications and developers. In addition, it interacts directly with Linux control groups [5], allowing Fastswap to enforce local memory allocations. Fastswap prevents head-of-line blocking by steering page fetches on the critical path to a

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387522

separate queue. Further, it reduces delays on the critical path by polling for completions of critical page operations and by offloading memory reclaim to a dedicated CPU. As a result, Fastswap achieves remote page access latencies of $< 5us$, enabling applications to access far memory at 10 Gbps with one thread, and 25 Gbps with multiple threads. Infiniswap [34] is the closest related work, and Fastswap's bandwidth is 1.51× (with one thread) and 2.54× (with multiple threads) higher than Infiniswap with disabled backup disk.

The second barrier lies in how one decides how to split each job's memory demands between local memory and far memory. The use of far memory is, to some extent, a bin-packing problem: how do you process a workload most quickly with a given amount of local memory on each server and a large pool of remote memory, and where each job must be assigned a total amount of memory (local and remote) that fully satisfies its requirements? To this end, we designed a **far memory-aware cluster scheduler** that leverages far memory to improve job throughput. When a new job arrives, the scheduler can place the job on a server that initially has insufficient available local memory to handle all jobs assigned to it. Our scheduler then reduces the local memory used by some of the existing jobs on that server, and uses far memory to ensure that all jobs have access to enough total memory. It is far from clear that such a strategy is beneficial, as using far memory inevitably slows down individual jobs (since accessing far memory is significantly slower than accessing local memory). However, using far memory can also enable more jobs to simultaneously run on a single server, albeit more slowly, which might increase overall throughput. We have studied this trade off extensively, and report on when the use of far memory increases overall throughput, and how this compares to merely increasing the amount of local memory. To the best of our knowledge, this is the first systematic exploration of these questions.

The combination of the improved swap system Fastswap and the cluster scheduler provides support for cluster-wide far memory, which we call CFM. While we do not have a large devoted cluster available to us, we used a cluster simulator (validated with runs on a real nine-node cluster) to explore what happens on a rack of 40 servers. We find that far memory is not a panacea; if the memory demands are substantially larger than available memory, then better performance is achieved by increasing the local memory per-server rather than by adding an equivalent amount of memory to a shared far memory server. However, we find that far memory provides significant benefits in two key scenarios on a single rack: (1) If the workloads are memory intensive (*i.e.,* memory availability rather than core availability is the bottleneck), converting a compute node into a far memory server can result (for the case we studied) in roughly 10% improved throughput compared to the original rack, even though *both rack configurations have the same amount of total memory.* (2) If an operator wishes to moderately increase memory capacity in a rack, adding memory to a memory server allows for finer granularity increases which still result in significant performance improvements, whereas upgrading the local memory in each server can only be done in much larger (and therefore more expensive) increments (as we discuss in the next section).

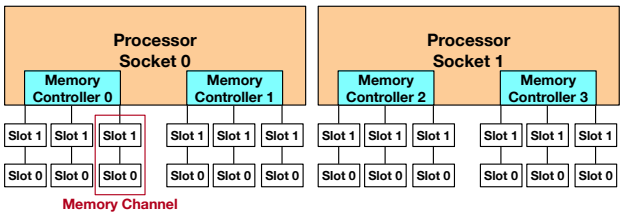

Figure 1. An example server platform with two sockets; each has two memory controllers, six memory channels, and twelve memory DIMM slots [17].

We have made available our Linux kernel modifications and drivers, far memory-aware scheduler, and simulator at https://github.com/clusterfarmem.

2 Context

As memory requirements for datacenter applications increase steadily, memory comprises an ever larger fraction of the costs of operating a datacenter [43, 44, 50]. As a result, cluster operators face difficult choices in how to provision memory in their datacenters. In this section we provide context about memory provisioning, and state the assumptions of the deployment scenarios we consider.

2.1 Memory Provisioning

It is important to remember that local memory can only be provisioned at a coarse granularity. As an example, consider the memory configuration of a recent 2-socket Intel platform [17]. As Figure 1 shows, each socket has 2 memory controllers, each memory controller has 3 channels, and each channel has 2 slots for memory modules, or DIMMs, making a total of 24 possible DIMM slots. DIMM sizes are typically powers of 2: 4 GB, 8 GB, 16 GB, and so on. However, one cannot efficiently use arbitrary numbers and combinations of DIMMs in the same server. Because of the way that processors interleave memory accesses, configurations that are "unbalanced" yield significantly lower memory bandwidth. For example, using 1 DIMM on each memory controller yields only 35% of the maximum system memory bandwidth [4, 17, 28].

The guidelines for balancing memory to achieve full memory bandwidth are the following: (1) all memory channels should have the same total memory capacity and (2) all memory controllers should have the same configuration of DIMMs (number and sizes of DIMMs [1]) [17]. Therefore, in Figure 1,

[1] In addition, all populated memory channels should have the same number of ranks. A rank is a block of memory and memory controllers interleave accesses to different ranks.

all slot 0s must have the same memory capacity and all slot 1s must have the same memory capacity.

Achieving balanced configurations limits the granularity at which memory can be initially provisioned, and also constrains the granularity at which memory can be upgraded if memory requirements change over time. For example, if we originally installed 192 GB of memory with twelve 16 GB DIMMs (one per channel), the smallest memory upgrade we could perform would be to add twelve 4 GB DIMMs (one per channel), representing a 25% increase in total memory capacity. If we then wanted to increase our memory capacity again at a later point, the smallest memory upgrade would involve *discarding* the 4 GB DIMMs and replacing them with 8 GB DIMMs (purchasing 40% more memory to increase our capacity by 20%). Worse, if we had originally installed 192 GB of memory by filling all twenty-four DIMMs in our platform with 8 GB DIMMs, the next memory upgrade would require us to discard 96 GB and buy twelve 16 GB DIMMs, resulting in 288 GB, a 50% upgrade from the original.

Thus, if you can only provision memory per-machine, the constraints on memory configurations require that you operate at significantly reduced memory bandwidth (which results in unacceptable performance) or else only increase your memory capacity at a very coarse granularity, yielding over or under-provisioning as memory demands change gradually over time [55, 61]. If we measure the granularity of memory upgrades in memory per core in the cluster, far memory can be upgraded at much finer granularity than local memory, despite obeying the same memory balancing constraints, because the added memory is shared by a much larger number of cores.

2.2 Deployment Scenarios

In this paper we are not primarily considering green-field deployments where an operator can assess their workload's memory requirements and then determine the most cost-effective way to meet them. Our results do shed some light on green-field scenarios – in particular showing that for memory-intensive workloads on a single rack, it is better to convert one compute server into a far memory server – but our main focus is on incrementally upgrading existing deployments. Based on what we have heard from operators (these are not operators of hyperscale datacenters, but more reflective of smaller commercial datacenters found in many businesses), their existing datacenters tend to have all DIMM slots currently filled. This makes economic sense, because the cost per unit of memory increases with DIMM capacity, so the cheapest way to provision a given amount of memory is to use all available DIMM slots.

Considering a single rack that has been provisioned this way, how should an operator respond when the memory demands of their workloads exceed what the rack can accommodate? As described above, upgrading the memory in all the servers in a rack can be done only at a coarse granularity (in terms of the ratio of memory to cores), whereas far memory can be added at a finer granularity. Our goal, in this paper, is to explore how these options compare.

To the best of our knowledge, datacenter operators ranging from small to medium prefer upgrading whole fleets at once. We do not consider the case where local memory is upgraded on a subset of the rack for two reasons: (1) we believe it would be challenging to manage memory-heterogeneous racks for operational reasons, and (2) we believe far memory would be better at avoiding memory over and under-provisioning due to static partitioning of memory across machines.

While far memory has the potential to improve cluster throughput, this comes at the cost of slower runtimes for individual jobs. As such, we believe that far memory is best suited for applications whose primary metric is job throughput, rather than customer-facing or latency-sensitive applications; that is our focus in this paper.

3 CFM Overview

The goal of CFM is to enable clusters to improve their end-to-end job throughput by leveraging far memory on dedicated memory servers. Though prior work has explored mechanisms to enable far memory and has demonstrated performance benefits of individual jobs when they swap (e.g., [34]), to the best of our knowledge, no prior work has demonstrated performance improvements with far memory at rack scale. We focus on improving end-to-end makespan, or the time it takes to finish executing a list of jobs. At a high level, CFM's approach (§3.1) bears similarities to prior work, but CFM overcomes several key challenges (§3.2) that make it difficult to reap cluster-scale benefits from far memory today.

3.1 Approach

In CFM, applications utilize far memory via swapping over RDMA. CFM enforces memory allocations using Linux control groups.

Swapping. Applications can leverage far memory in broadly two ways: transparently (without application modifications), or via explicit and potentially custom APIs [1, 23, 54, 59]. Though specialization has the potential to offer better performance, large-scale cluster operators have claimed that specialization is impractical [43], as their clusters execute thousands of heterogeneous workloads, and porting them to a different memory hierarchy would take significant effort. Instead, similar to Infiniswap [34], CFM realizes far memory with swapping, an existing mechanism that extends virtual memory beyond physically available memory. When a CPU accesses a memory address in a page that is not present in physical memory, a page fault is raised and the page fault handler transparently fetches the contents of the page from swap space into local memory. Traditionally swap space resided on disk

and the resulting millisecond-scale access latencies induced large and poorly understood performance overheads on workloads. However, swapping itself does not fundamentally entail millisecond-scale latencies, and with today's microsecond-scale network latencies, swapping to far memory over the network has the potential to yield good performance.

Cgroups. CFM enforces per-job limits on local memory consumption using Linux control groups (cgroups) [5]. Cgroups control the amount of physical memory allocated to a group of processes and CFM uses the swap system to keep the excess in far memory.

RDMA. CFM leverages RDMA for low-latency access to memory on remote servers. CFM uses one-sided *read* and *write* operations, which enable access to memory without using the remote CPU. In general, RDMA operations are submitted to local *queue pairs* and are then processed by the local RDMA NIC. Once an operation has completed, the NIC posts a *completion* to a *completion queue*; completion queues can be configured to raise interrupts when completions arrive, or to remain silent with the expectation that they will be polled. Traditionally RDMA bypasses both the remote and local operating systems, but RDMA also offers a kernel API for drivers to use; CFM leverages this API for swapping pages over the network.

3.2 Challenges and Contributions

There are two main challenges to realizing cluster-scale benefits from far memory: enabling far memory to be swapped in quickly (§3.2.1) and deciding how to schedule jobs across local and far memory (§3.2.2).

3.2.1 Fast Swapping. RDMA swap devices have been explored in prior works such as Infiniswap [34] and HPBD [47]. However, these approaches are unable to sustain the high performance required by applications today, for three main reasons:

1. To hide the I/O latency of future page faults, operating systems typically implement page prefetching by fetching several pages on each page fault. Unfortunately, in Linux the *faulted page*—the page currently required by the application—may lie anywhere in the aligned window of pages to be prefetched. Existing systems fetch all pages using a single queue pair per CPU (or worse, a single queue pair for the whole swap system [47]), so the faulted page may queue behind prefetch pages. Processing each page to fetch can take a few microseconds due to memory allocation, so with Linux's default prefetch window size of 8, head-of-line blocking may delay fault handling by tens of microseconds.
2. In existing systems that swap over RDMA, the CPU is notified that an RDMA operation (e.g., a read of a remote page) has completed via interrupts. This interrupt-handling occurs on the critical path—before the page fault handler is able to return to the application—and can add 10 μs or more [13] to page fault handling.
3. After the contents of a faulted page are read into local memory, the operating system charges the new page to its cgroup by increasing its memory counter. If the cgroup memory limit is exceeded, excess pages need to be reclaimed. In contrast to system-wide reclaim in Linux, memory reclaim in cgroups is always done directly, that is, before leaving the page fault handler and returning to the application. Thus the entire process of reclamation (finding pages to reclaim, writing them to the swap device, and returning the pages to the kernel for reuse) delays page fault resolution.

CFM introduces a faster swapping system called Fastswap (§4) that overcomes all three of these challenges, enabling CFM to achieve lower latency and higher throughput for remote swap than existing systems such as Infiniswap (§6.4).

3.2.2 Cluster Scheduling. Many existing schedulers enable efficient sharing of cluster resources by scheduling jobs across cores, memory, and other resources [31–33, 36, 51, 56, 61]. However, existing schedulers do not consider far memory; that is, they do not provide support for scheduling jobs whose memory can be dynamically split across local and far memory and they do not specify how to best allocate local memory across multiple jobs sharing the same machine. CFM proposes a centralized far memory-aware scheduler (§5) that considers far memory when assigning jobs to machines and decides how to partition local memory across different jobs in order to optimize makespan.

4 Fastswap

In this section we describe Fastswap, our RDMA swap system. Figure 2 shows its overall architecture and how it improves upon existing components of the operating system. While previous research efforts [30, 34, 47] simply expose their RDMA backend as a swap device, we found that enabling higher swapping performance required modifications to the page fault handler, swap system, and the cgroup memory controller. We implemented Fastswap by modifying approximately 300 lines of kernel code, and with a new device driver in 1200 lines of code for Linux 4.11.

Improving paging performance is challenging. While many systems focus on making improvements at millisecond time scales [35, 46], our system strives to enable microsecond-scale swapping. Most of the mechanisms we discuss in this section occur while program execution is *paused*. Therefore, every microsecond we save is a microsecond of compute time given back to the application.

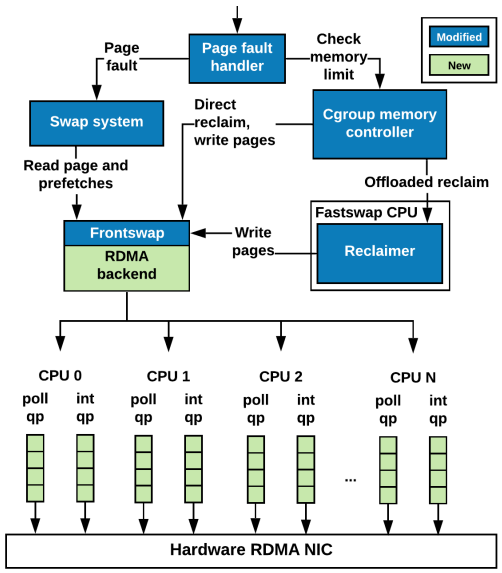

Figure 2. Architecture of Fastswap.

4.1 RDMA Backend

In Fastswap, the operating system interacts with the RDMA NIC using the RDMA backend. As shown in Figure 2, the backend is used by all swap operation types: page faults, prefetches, and memory reclaim. While prior research has exposed an RDMA backend as a block device [34, 47], Fastswap uses the *Frontswap* interface [3]. Frontswap is designed for swapping at page granularity rather than supporting general block I/O operations, and strives to minimize context-switches to other tasks while swap operations complete.

Queue pairs. RDMA requests in a given queue pair are processed in-order by NIC Processing Units [39]. If different classes of swap operations share a queue, critical operations—*e.g.*, reads for faulted pages and writes for evicted pages—will queue behind less urgent prefetch reads. Fastswap avoids this head-of-line blocking by using two RDMA queue pairs per CPU, one for operations on the critical path and one for prefetches. Separating these operations into two queue pairs enables Fastswap to handle their completions differently. Our RDMA backend configures interrupt completions for prefetches, and disables them for critical operations (indicated by *int* and *poll* in Figure 2); Fastswap polls for completions of critical operations instead.

Frontswap interface. Frontswap assumes its operations complete synchronously [3], that is, control of execution returns to the swap system only after the Frontswap operation has completed. Therefore, it provides no mechanism to distinguish between operations that are on the critical path and those that are not. Thus, Fastswap enhances the Frontswap interface to distinguish between critical and non-critical operations, enabling the RDMA backend to steer requests to their appropriate queue pair. In our modified Frontswap interface, both types of operations return immediately after initiating

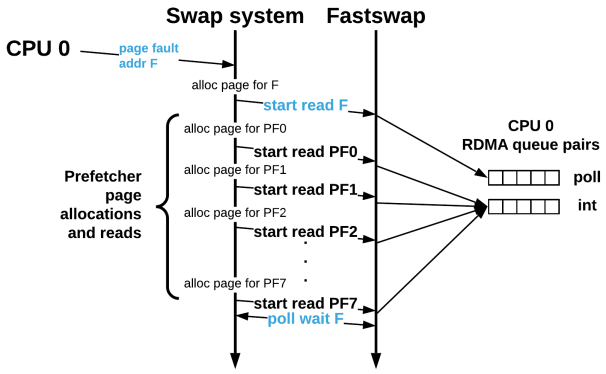

Figure 3. Page fault and prefetching with Fastswap.

their RDMA request. The modified swap system then polls for completion of critical path operations, while non-critical path operations trigger interrupts on completion.

4.2 Page Fault Handler

Fastswap modifies the page fault handler in two key ways. First, it instructs the swap system to handle faulted pages and prefetched pages differently, as described above. Second, Fastswap modifies the swap system to *first* read the faulted page, followed by the remaining pages within the prefetch window (a related approach is proposed in [26]). After issuing all reads, Fastswap poll waits for the faulted read to complete. By issuing the faulted read first, we overlap the latency of allocating physical pages for prefetch reads and the latency of issuing the prefetch RDMA reads, with the RDMA read for the faulted page. Figure 3 shows how Fastswap services a page fault and associated prefetches.

Handling faulted pages and prefetched pages separately minimizes the cost of missed prefetches. For example, assume page fault 1 occurs on address F_1, with an associated set of prefetch pages P_1. Our swap system will issue reads for F_1 and P_1, and poll until the read for F_1 completes. At this point, the page fault handler will return to user space. Then, suppose page fault 2 occurs on address F_2 where $F_2 \notin P_1$ (*i.e.,* a prefetch miss). Fastswap can fetch F_2 without waiting for any page in P_1, whereas previous systems would need to wait for all pages in P_1 before the F_2 read could finish [34, 47].

4.3 Memory Reclaim

We have covered how the Fastswap swap system brings pages from far memory into local memory. Now we describe how Fastswap reclaims memory so that processes do not use more local memory than their allowed share. Fastswap moves reclamation off the critical path of page fault handling by modifying the cgroup memory controller.

In general, memory reclaim is needed when memory in a cgroup grows beyond its allowed limit, or conversely, when a cgroup's memory limit shrinks. In Fastswap, memory in

a cgroup increases when page faults bring pages from far memory into local memory, or when a process allocates additional memory. Meanwhile, memory limits shrink when our far memory-aware scheduler chooses to carve out memory to fit additional processes (see §5).

Traditionally, after reading a faulted page, the memory controller charges the page to its cgroup. Then, the controller checks if the cgroup has more memory than its allowed share. If there are excess pages, they are *directly reclaimed* and possibly evicted to far memory. Direct reclaim takes place in the context of the page fault handler, so it prevents the CPU from returning to user space and continuing workload execution.

As we show in §6.4.3, memory reclaim in Linux is surprisingly expensive, consuming 62-85% of the kernel time when our applications have 50% of their memory in far memory. To reduce these costs, whenever a node is using far memory, our modified memory controller offloads memory reclaim to a dedicated reclamation CPU (Figure 2); we call this process *offloaded reclaim*. Offloading memory reclaim allows the CPU that caused a page fault to return to user space without spending time on direct reclaim. Recent efforts have used a similar approach for offloading cold memory compression [43] and packet processing [22, 52] to dedicated CPUs.

Offloaded reclaim is not suitable for all situations, *e.g.*, to reclaim pages in response to large memory allocations, or in response to a large limit shrink. In these situations, offloaded reclaim can become a bottleneck because the reclaimer is shared across CPUs.

To prevent cgroups from significantly exceeding their memory allocation, Fastswap gives each cgroup a small threshold α of memory above its limit. When a cgroup first reaches its memory limit, the memory controller requests offloaded reclaim. If the reclaimer is busy and cannot service reclamation requests fast enough, memory in the requesting cgroup will keep increasing. Once the cgroup exceeds its memory limit by α, page fault handlers for the cgroup will perform direct reclaim as well. This guarantees that the cgroup does not exceed its limit by more than α. In our implementation we use α=8MB. If a node is not using far memory, the reclaimer is idle, so the CPU can be used to execute jobs.

Regardless of whether memory reclaim is direct or offloaded, when evicting pages to far memory, we poll for their completions. A page can be fully reclaimed only after its write to far memory finishes. At this point, the cgroup memory allocation decreases and the kernel can reuse the page. Since memory reclaim is done in batches, using interrupts for writes would delay observable reclaim and could cause more page reclaims than required. With polling, the cgroup memory counter decreases immediately after the write completes.

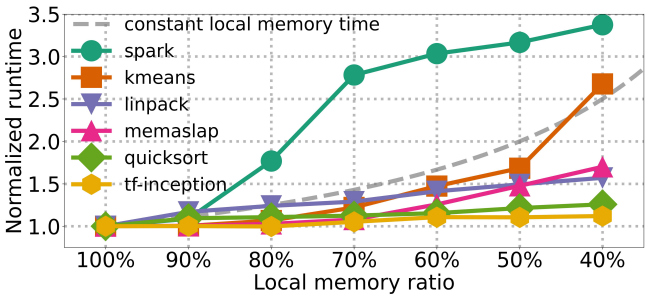

Figure 4. Performance degradation of applications from Table 1 using Fastswap. The constant local memory time line depicts A=B (§5.2).

5 Far Memory-Aware Scheduler

In this section we describe how CFM makes scheduling decisions for a cluster equipped with far memory. Our scheduler uses bin-packing to allocate memory between jobs, and gains greater flexibility by allowing jobs to use far memory in addition to local memory. Intuitively, this can improve job throughput by allowing each node to fit additional jobs when memory is the constraining resource. However, the use of far memory slows down individual jobs, and as a result it is unclear how this impacts the overall makespan.

We define the memory request mem_i of job i to be the maximum it uses during execution; if allocated mem_i of memory, job i would not incur any hard page faults. Given a set of jobs and their CPU and memory requirements, there is a maximum number of jobs we can fit onto a single server. By using cgroups and CFM, we can rebalance the local memory that jobs use in a node, and free up enough local memory to fit additional jobs. When a cgroup is shrunk, pages are evicted to far memory such that memory of the cgroup does not exceed the new limit. In general, the cost of shrinking is the slowdown jobs experience, and the network bandwidth and latency to move memory from local to far memory.

5.1 Job Degradation Profiles

The performance degradation applications experience when they trade local memory for far memory is application dependent (see §6.1.2 for details on the applications we use). Figure 4 shows how the runtime of several jobs (each normalized by its runtime with 100% local memory) increases as we decrease the local memory ratio, or the fraction of the job's memory that is local. Some applications such as tensorflow-inception experience very little slow down (at most 10.5%) when using far memory, while others experience significant slowdowns (*e.g.*, spark's runtime triples when using 40% far memory).

Therefore, a scheduler that uses far memory cannot treat every job the same—it needs additional information that allows it to model the job slowdown in order to optimize workload makespan. Thus, for each application that we use in the rest of

the paper, we create a *degradation profile* [2] that estimates the runtime $f_i(r_i)$ at different local memory ratios r_i. To create a degradation profile, we measure the application's runtime at several discrete local memory ratios using Fastswap, and then use polynomial fitting to create a continuous function that the memory policy uses (explained in the next subsection).

Using job profiles has limitations; in particular, applications must finish so their profiles can be computed, and applications must have similar performance degradation when using far memory across different executions. It may be possible to use page fault frequencies to model job slowdown instead of pre-computed profiles; we leave exploring this to future work.

5.2 Far Memory Scheduling Policies

Our scheduler is simple and follows conventional designs [61], except when dealing with memory. When new jobs arrive, they are added to a pending queue which keeps jobs in arrival order. Whenever the pending queue is not empty, the scheduler tries to assign jobs to nodes in the job-arrival order. For each job, the scheduler iterates through all nodes in order to find one that has a sufficient number of cores and memory available for the job. We iterate through the nodes in random order to improve the average runtime for finding such a node, and the job is placed on the first such node found. If no node has sufficient resources to execute a job, we leave it in the pending queue. In this work we do not consider job preemption.

When scheduling jobs, the scheduler needs to make two decisions. First, in the loop above it must decide if a job "fits" on a node (*i.e.,* perform job admission control); in order to do so, we rely on a `fit` function provided by the memory policy. The `fit` function begins by checking if sufficient cores are available to execute the job; if not, we declare that the job does not fit on the node. If sufficient cores are available, we use a heuristic to determine whether sufficient local memory can be made available at the node, and enough free far memory remains to execute the job. If so, we say that the job fits on the node; otherwise, we say that it does not.

Second, once a node with sufficient resources has been found, we need to decide how much local memory to give the job. This is done by the `rebalance` function provided by the memory policy. The `rebalance` function revises the memory allocations of jobs executing on the node, so it is called *before a new job is started* to free enough local memory for the new job, and *after a job finishes*, in order to distribute local memory back across the remaining jobs. `rebalance` does not use far memory unless it needs to.

Next, we discuss several memory policies that we considered in terms of the `fit` and `rebalance` functions they provide.

[2] We note that our degradation profile is similar to Miss Ratio Curves [63], except in MRC the y-axis is miss ratio, whereas in our profile it is job execution slowdown.

Uniform policy. When a set of jobs on a machine requires more memory than is available locally, this policy shrinks all jobs uniformly up to a minimum ratio α. For example, if $\alpha = 0.75$, then the policy will trade up to 25% of local memory for far memory on every job. This policy uses this minimum ratio both to determine if jobs fit and to rebalance memory allocations. Although simple, this policy does not take into account the fact that different jobs slowdown differently when they are shrunk to the same ratio. In addition, the same ratio will mean different amounts of far memory depending on how much memory each job uses.

Variable policy. This policy improves upon uniform by allowing per-job minimum ratios. We chose minimum ratios for jobs that correspond to a 20% slowdown—we empirically determined 20% to be a good trade off between job slowdown and improved makespan. The `fit` function returns true if 1) there is enough memory on the machine for all jobs, including the incoming job, to have at least the amount of local memory specified by their minimum ratio, and 2) the cluster has sufficient far memory for the residuals. The `rebalance` function adjusts memory allocations by reducing local memory proportionally for each job according to its minimum ratio. Therefore, the minimum ratio of each job determines its friendliness for using far memory. Rebalancing jobs proportionally based on their minimum ratios requires users to define it for each job. Since this policy scales local memory linearly for each job, it performs best if performance degradation is also linear up to the minimum ratio.

Memory-time policy. Using insights gained from the previous policies, we designed a policy that directly captures the fact that jobs that use far memory experience nonlinear slowdowns. As a result, rather than relying on manually specified minimum ratios, this policy makes use of the memory-time product, which we explain next, to determine the best local memory ratio for each job in a node when `rebalance` is called.

Given a set of memory-constrained jobs, the asymptotic makespan for these jobs is given by [32, 33, 61]:

$$makespan \approx \frac{memorytime}{local\_mem \cdot utilization}$$

where *memorytime* is the sum of all the jobs' memory requirements multiplied by their runtime, *local_mem* is the total available local memory in the cluster, and *utilization* is the average utilization of local memory in the cluster. Intuitively, the product of memory requirement and runtime captures how much memory a job consumes during its execution. Without far memory, *memorytime* is fixed because the amount of local memory a job uses and its duration are both fixed. The total local memory in the cluster is similarly fixed. Therefore, previous research on scheduling to lower makespan, or improve throughput, could only focus on increasing the denominator by improving *utilization* [32].

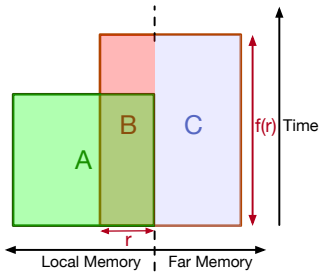

Figure 5. How a job can reduce its local memory consumption by using far memory. A is the original memory-time product when no far memory is used. B+C is the new memory-time product, where B is the local portion, and C is the far portion of the product. r is the local memory ratio of the job.

With far memory, we can decrease local *memorytime* and increase *utilization* to further improve makespan. Since we use far memory only when local memory is fully utilized, *utilization* is commonly very high. In addition, *memorytime* is no longer fixed at $\sum_{i=1}^{N} mem_i \cdot f_i(1)$ but is instead flexible:

$$memorytime = \sum_{i=1}^{N} mem_i \cdot r_i \cdot f_i(r_i)$$

As we increase far memory usage, the product $r_i \cdot f_i(r_i)$ depends on how gracefully a job's performance degrades. Figure 5 conceptually shows A, the original memory-time product for a job using only local memory, and B+C, its new memory-time product when using far memory (*i.e.,* B is the local memory-time product while C is the far memory-time product). As long as the area of B is less than A, a job's local *memorytime* can be reduced by trading some of its local memory for remote memory. When this is not the case, using additional far memory would increase a job's local memory-time, so our scheduler must not reduce a job's memory ratio below this point. As illustrated in Figure 4 with the *constant local memory-time* curve, for many jobs the slowdown is graceful enough that they can be shrunk significantly without reaching the point where A and B have the same area. For example, we could shrink local memory at least 60% for all six workloads except for spark and linpack. Therefore, our memory-time policy can reduce local *memorytime* by finding appropriate local memory ratios for each job. For example, for $r_i = 0.5$, if $f_i(0.5) < 2 \cdot f_i(1)$, then we reduce the local memory-time product because we save half of the job's local memory while incurring less than twice the slowdown.

We turn this insight into a better local memory rebalancing policy by considering three factors. First, to optimize for makespan, we should pick local memory ratios for each job to minimize the sum of their local memory-time products (i.e., the sum of the A's of existing jobs in the machine). If we had unlimited far memory, this would be sufficient. However, given that we have a limited amount of far memory that is shared by several machines we also need to ensure efficient allocation of far memory. Unfortunately, optimizing memory allocations over all local and far memory is at least APX-Hard [64] and thus not feasible in our setting.

To resolve this, CFM optimizes each node independently by choosing local memory ratios for jobs such that they maximize the ratio between the savings in local memory-time products and the increase in far memory usage. This avoids global optimization while still using local and far memory efficiently. We do this by solving the following optimization problem:

$$\begin{aligned}
\underset{r_i : i=1,\ldots,N}{\text{maximize}} \quad & \frac{A-B}{C} \\
\text{subject to} \quad & \sum_{i=1}^{N} mem_i \cdot r_i = local_mem \\
A-B \quad & = \sum_{i=1}^{N} mem_i \cdot (1-p_i) \cdot f_i(1) - mem_i \cdot (1-p_i) \cdot r_i \cdot f_i(r_i) \\
C \quad & = \sum_{i=1}^{N} mem_i \cdot (1-p_i) \cdot (1-r_i) \cdot f_i(r_i)
\end{aligned}$$

where p_i is a ratio between 0 and 1 that represents the progress of this job according to its profile, and $(1-p_i) \cdot f_i(r_i)$ is the remaining run time for job i using local memory ratio r_i. Therefore $A-B$ is the total local memory-time product saving, C is the total far memory-time product, and the equality constraint ensures that local memory is fully utilized.

Each node tracks the progress of its own jobs (updating each p_i when the local memory ratios change) and solves the optimization problem when `rebalance` is executed. The dimension of this optimization is the number of jobs running on the node, and it converges in a small number of iterations using conventional optimization tools; *i.e.,* we found that SciPy [62] was fast enough for our scheduler and our simulator.

The memory-time policy does not prescribe how to determine whether or not a job fits on a machine; *i.e.,* it is not involved in job admission. For simplicity, we use the same `fit` function as the variable policy. We leave more advanced admission control schemes (*e.g.*, [32]) for future work.

Although CFM does not primarily target latency-sensitive production workloads, it is flexible enough to exempt these from using far memory. We can do so by specifying a minimum admission memory ratio of 100% for the job in the `fit` function, and by giving the job a profile where $f(1) = \text{runtime}$, and $f(\text{anything else}) = \infty$. Note that this way, other jobs can still trade local memory for far in order to fit the far memory-exempt job. We leave for future work analysis and evaluation of these scenarios; in this paper we assume every workload can be slowed down.

5.3 Scheduler Implementation

Our scheduler is comprised of a central scheduler and a per-node daemon. We implemented both in 1500 lines of Python. The central scheduler implements the design outlined in §5.2,

and uses gRPC to communicate with all daemons. When the scheduler dispatches jobs to a daemon, the daemon creates a cgroup with a memory limit for it— the memory limit is defined by the memory policy being used. Often, the memory limit is smaller than the amount of memory the job needs, which triggers far memory usage. The scheduler can be configured to not use far memory at all; in this case, the scheduler propagates this configuration to the daemons so they will not use far memory.

6 Evaluation

Our evaluation of CFM focuses on three main questions:

1. How does CFM perform in a real testbed? (§6.2)
2. What are the benefits of far memory and when should one use far memory instead of local memory? (§6.3)
3. How do CFM's individual components contribute to its overall performance? (§6.4)

6.1 Experimental Setup

We evaluate CFM on a small testbed rack and in simulation at rack scale using a cluster simulator.

Testbed rack. Our testbed consists of 14 machines; we use 9 as compute nodes, 1 as the scheduler, and up to 4 as memory servers. Each machine has an 8-core Intel Xeon E5-1680 v2 CPU, 32 GB of memory and a 40 Gbps Mellanox ConnectX-3 NIC. We use one hyperthread on each core and disable TurboBoost and CPU frequency scaling in order to reduce variability. Each machine runs Ubuntu 16.04 with Linux kernel version 4.11. Each job runs in its own cgroup; memory and core allocations are decided by the CFM scheduler.

Cluster simulator. We implemented our own cluster simulator in 2200 lines of Python. It implements the same `fit` and `rebalance` functions as our testbed implementation. Our simulator takes as input a degradation profile for each job produced by using Fastswap (§5.1), and uses these to determine how long each job takes to complete, given the memory resources allocated to it. If the cluster scheduler changes a job's local memory allocation during its execution, our simulator adjusts the job's remaining runtime accordingly. Since our profiles are generated when applications are executed individually, our simulation results represent a best case performance analysis of using far memory. We evaluate the accuracy of our simulator in §6.3. Our simulated cluster consists of 1 rack of 40 machines, each with 48 cores and 192 GB of memory.

6.1.1 Evaluated Systems. We evaluate four systems. First, we evaluate a baseline cluster that uses only local memory, denoted by **NoFar**. To understand the benefits of additional local memory, we use **NoFar (+X%)** to denote the configuration in which each machine in the baseline cluster has been augmented with X% additional local memory. Second, we evaluate **Infiniswap** [34], a system that enables applications to leverage remote memory on other servers (rather that in a dedicated memory server), via RDMA. Infiniswap requires writing to disk to correctly function in a cluster. However, we found that writing to disk severely degrades its performance, so we include a modified version where we disabled writing to disk. Infiniswap does not specify how to schedule jobs, so we only evaluate it in single-job experiments. Third, we implemented a Fastswap **DRAM** backend for comparison and to understand how swapping operations are impacted by RDMA performance.

Benchmark	Memory (GB)	# cpus
linpack	1.56	4
quicksort	8.05	1
kmeans	4.73	1
tensorflow-inception	2.07	2
memcached	12.00	2
spark-pagerank	4.29	3

Table 1. Applications that comprise our workloads.

Finally, we evaluate CFM using the memory-time policy unless specified otherwise. We use **Far (+X%)** to denote the configuration in which the cluster runs CFM and has been provisioned with far memory such that the cluster's total memory is X% more than in NoFar. When evaluating any Far configuration, we use one fewer server per rack than in NoFar to ensure that there is available space in the rack for a memory server to support far memory.[3] In addition, when far memory is in use, CFM dedicates at least one core per server to running the reclaimer. We found that one core was sufficient for the reclaimer on our 8-core testbed machines; based on its utilization (39.5% on average), we estimate that our larger simulated 48-core machines require 3 reclaimer cores and thus we give three fewer cores per simulated server for CFM.

6.1.2 Jobs and Workloads. In the experiments that follow, a *job* can be any of the applications described in Table 1. We focus on applications that can benefit from cluster throughput improvements such as analytics applications. We now describe each application in more detail. linpack is a linear algebra performance benchmark, and we use an Intel provided binary which we limit to use 4 CPUs [7]. quicksort uses the C++ standard library to sort 8GB of integers. kmeans uses sklearn to classify 15M samples [9]. tensorflow-inception does inference on an inception reference implementation for benchmarking [10]. memcached uses memaslap to SET 30M keys and then query 100M keys using the ETC distribution (*i.e.,* 5% SET, 95% GET), while memcached is pinned to another CPU [12, 34]; memcached could be used as a parameter server in this context. spark-pagerank uses a dataset of 685K nodes and 7.6M edges [45].

[3] Because the CFM configurations use one fewer server, Far (+0%) has one server's worth of far memory, to yield the same total rack memory as NoFar.

m2c	NOFAR	FAR (+0%)	FAR (+11%)	FAR (+33%)
1.0	1.00	1.05	1.04	1.07
1.2	1.00	1.12	1.08	1.10
1.4	1.00	1.07	1.12	1.11
1.6	1.00	1.15	1.21	1.28

Table 2. Makespan *improvement* in testbed normalized to NOFAR (*i.e.,* 9 node cluster without far memory).

A *workload* is a list of 6000 mixed jobs with uniformly random arrivals. Every workload has at least one instance of each job from Table 1. We characterize workloads using two properties: **m2c** and **packability**.[4] m2c captures how a workload's demand for memory relative to compute compares to that of the underlying cluster. For a workload W with N jobs and a cluster C with C_{cpu} cores and C_{mem} GB of memory, we define $m2c$ as:

$$m2c(W,C) = \frac{\sum_{j=1}^{N} mem_j \cdot duration_j}{\sum_{j=1}^{N} cpu_j \cdot duration_j} \cdot \frac{C_{cpu}}{C_{mem}}$$

For example, a workload that consists of jobs that require the full memory but only half the cores of any machine in the cluster has an m2c of 2. To produce workloads with a given range of m2c values, we randomly generate many workloads by varying the ratio of each job in the workload, and then select those that have an m2c within the given range.

Packability captures how easily a workload's jobs can be scheduled in a cluster without using far memory. We define the packability metric for a workload and cluster as the makespan achieved when all of a cluster's resources (memory and cores) are pooled into one large server (eliminating any resource fragmentation),[5] divided by the makespan achieved in the default NOFAR configuration. Thus, a packability of close to one indicates that a workload suffers little from resource fragmentation; as fragmentation increases, packability decreases.[6]

6.2 Testbed Performance

We use our testbed to evaluate how far memory behaves in real executions using our far memory-aware scheduler (§5) and Fastswap (§4). In this subsection, FAR (+0%) is an 8-node rack with 32 GB of far memory, and NOFAR is a *9-node* rack without far memory. We further explore the performance of FAR by adding 32 GB and 96 GB of memory to the far memory server, yielding the FAR (+11%) and FAR (+33%) configurations. We evaluate workloads that have m2c between 1.0 and 1.6, with a granularity of 0.2; in each case, we picked a single workload with an m2c close to the prescribed value.

[4]Both of these properties depend on the infrastructure on which the workload is run. Since we use a particular rack configuration as a baseline, we refer to these two measures without specifically mentioning the infrastructure.
[5]This pooling approach mimics the upper bound described in [32].
[6]Note that packability is defined with respect to a certain cluster scheduling algorithm; we assume jobs are scheduled as described in §5.2.

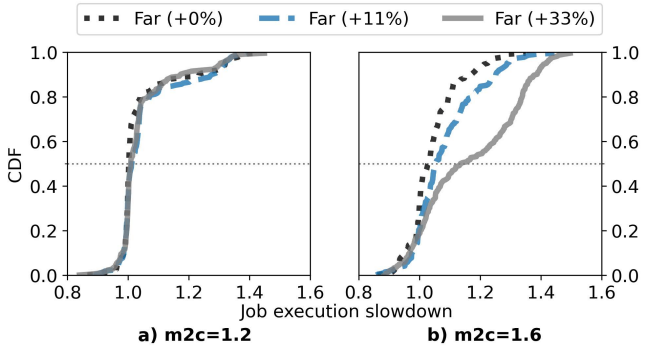

Figure 6. Job execution time slowdown when using far memory, relative to execution time in NOFAR, in our testbed. Left shows workloads with $m2c = 1.2$ and right shows workloads with $m2c = 1.6$.

Table 2 shows the makespan improvement over NOFAR. We make three key observations from these results. First, for memory-constrained workloads, using far memory reduces makespan. Most notably, with the same amount of total memory, FAR (+0%) outperforms NOFAR in throughput by 5-15%. Second, far memory helps more when workloads have a higher m2c. In the extreme case, FAR (+33%) outperforms NOFAR by 28%. Third, additional far memory does not always lead to better performance. The reasons are two-fold: (1) Additional memory can help only when memory is the constraining resource. Therefore, when workloads have a low m2c, a small amount of far memory can sufficiently mitigate contention over memory, so additional far memory provides minimal benefits. (2) As we observe from FAR (+11%) and FAR (+33%) for an m2c of 1.4, adding more far memory can slightly degrade performance. We believe this is due to our overeager admission policy; *i.e.,* memory-time uses the variable policy for admission control (see §5.2). In other words, it is possible that a job is admitted using a large fraction of far memory, while a better decision would have been to wait and admit the job later using less far memory, resulting in lighter slowdown. We leave the design of a better admission control policy as future work.

While CFM improves cluster throughput, it does increase the execution time of individual jobs. Figure 6 shows the CDF of execution time slowdown when using far memory relative to the same job's execution time in NOFAR, in our testbed. When the workloads are lightly memory constrained (*i.e.,* $m2c = 1.2$), using far memory slows down jobs by 0.2%–1.1% at the median, and by 35%–37% at the 99th percentile. As the workloads become more constrained by available memory, our scheduler can use more far memory at the expense of job execution time. As such, when $m2c = 1.6$, using far memory slows down jobs by 2%–13.4% at the median, and 28%–45% at the 99th percentile.

6.3 Rack-scale Evaluation

We use our simulator to evaluate the benefits of far memory for a full rack of 40 machines.

6.3.1 Simulation Validation. To validate that our simulator accurately emulates the behavior of our testbed experiment, we simulated the executions presented in Table 2. We found that our simulated makespans ranged from 9% less than to 3% more than the actual makespans measured in the testbed.

6.3.2 Benefits of Far Memory. We quantify the benefits of far memory in a rack by simulating many workloads with different amounts of far memory. Each workload has 6000 mixed jobs from Table 1, and lasts for an average of 107 minutes of simulated time. For this experiment, we consider three different ranges of m2c values: [0.75, 0.85], [1.15, 1.25] and [1.55,1.65]; each range includes 260 different workloads and each makespan is the average of 15 trials. Every workload is simulated without far memory (*i.e.,* NOFAR) and with different amounts of added far memory (*i.e.,* FAR (+X%)); we compute the percent improvement in makespan, relative to NOFAR, for each far memory configuration.

In Figure 7, each CDF shows the distribution of workload makespan improvements for a given m2c range and amount of far memory. Figure 7a demonstrates that for workloads that are on average CPU-constrained rather than memory-constrained (m2c values between 0.75 and 0.85), adding far memory imposes a small performance penalty of 6% at the median and 7% at the 99th percentile. This is because with far memory the rack has one fewer server, and if far memory is used, each server dedicates 3 cores to handle memory reclamation leaving fewer cores for jobs.

For workloads that are on average slightly or significantly memory-constrained, adding far memory provides benefits for all workloads and amounts of far memory. Even for the configuration with the smallest amount of far memory (*i.e.,* 192 GB, or FAR (+0%)), which replaces a compute server with a memory server with an equivalent amount of far memory, makespan improves at the median by 3% and 11%, as Figure 7b and 7c show. Even though this configuration has *the same total amount of memory and many fewer cores* per rack than NOFAR, it achieves lower makespans because the presence of a sharable far memory allows jobs to be packed onto machines that have available cores but not memory, thereby enabling resources to be used more efficiently.

Adding additional far memory continues to provide further benefits. For m2c values in [1.15,1.25], improvements relative to NOFAR plateau at a median of 12% with FAR (+17.5%); for m2c values in [1.55,1.65] improvements continue until our largest far memory point FAR (+37.5%) with a median improvement of 47%. We expect improvements to plateau because our memory policy prevents jobs from being admitted if doing so would require shrinking any job beyond its minimum ratio. Further, when a previously memory-constrained node has enough far memory, memory stops being the constraining resource, CPUs become the limiting resource, and additional far memory provides no benefit.

While we explicitly controlled the m2c ratio in our workloads, packability (as defined in §6.1) arises from the difficulty of packing jobs from each workload in the rack. Figure 8 shows improvement in makespan for FAR (+17.5%), relative to NOFAR, for a wide range of m2c values ([1.15,1.65]). It illustrates the effect that packability has on throughput improvement when far memory is available. As packability decreases, workloads are harder to pack in NOFAR, so far memory is able to provide more benefit; the median improvement at (0.93,0.96] is 12.5% while the median improvement at (0.85,0.88] is 25.8%. Thus we observe that both m2c and packability can significantly impact the makespan improvements a workload achieves from far memory.

6.3.3 Adding Far Memory vs. Local Memory. We now evaluate system performance when adding far memory compared to adding memory locally to each machine, *i.e.,* NOFAR (+X%). We aim to answer the question "If I purchase X GB of memory, should I add $\frac{X}{N}$ GB to each machine or X GB to a shared memory server?" We again consider different ranges of m2c values. For each range and memory configuration, Figure 9 plots the median percent improvement in makespan, relative to NOFAR.

When we add memory locally to each server, we can only do so in a few discrete amounts dictated by the current memory configuration and standard DIMM sizes (§2.1). For our simulated rack where each machine has 192 GB of memory, we consider two ways of initially provisioning each machine: (1) with 12×16 GB DIMMs, and (2) with 24×8 GB DIMMs. The smallest feasible upgrade for case 1 involves purchasing 48 GB per server, for an additional rack memory of 1.92 TB and a total rack memory of 9.6 TB. The smallest feasible upgrade for case 2 requires purchasing 12×16 GB=192 GB and *discarding* 12×8 GB=96 GB per machine, resulting in an additional 3.84 TB of memory in the rack and a total purchased memory of 15.36 TB (including the discarded memory). We simulate these two upgrade options and illustrate them in Figure 9 with the stars at x=9.6 TB (NOFAR (+25%)) and x=15.36 TB (NOFAR (+50%)), respectively.

We see that overall it is better to add memory locally on every machine, rather than add the same total amount of memory as far memory, in terms of the makespan. For example, when $m2c > 1.15$ and 9.6 TB of memory are given to the rack, NOFAR (+25%) has a makespan that is lower by 2.5% on average across the different m2c ratios, relative to FAR (+25%) (see $x =$ 9.6 TB in Figure 9). Similarly, when we add 3.84 TB to the rack, doing so locally results in a makespan that is lower by 9.7% on average, relative to FAR (+50%), shown by the right-most set of stars vs. the right-most dots. However, with far memory we

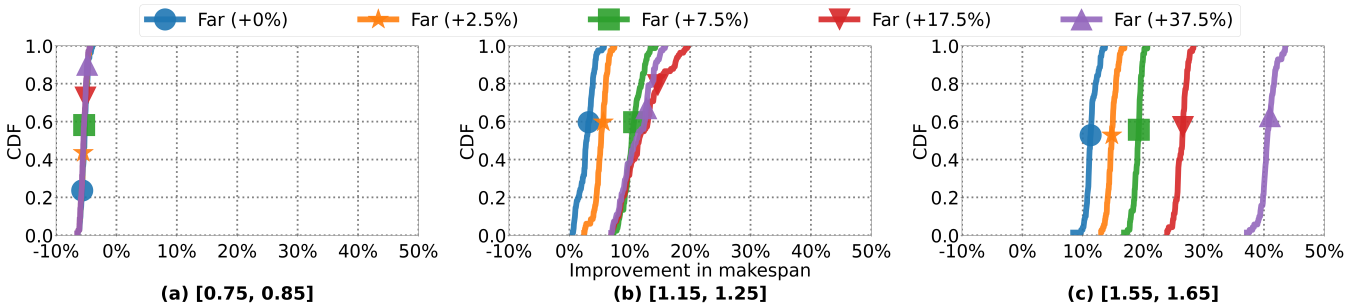

Figure 7. The percent improvement in workload makespan, relative to the NOFAR configuration, for workloads with three different ratios of memory to compute (m2c).

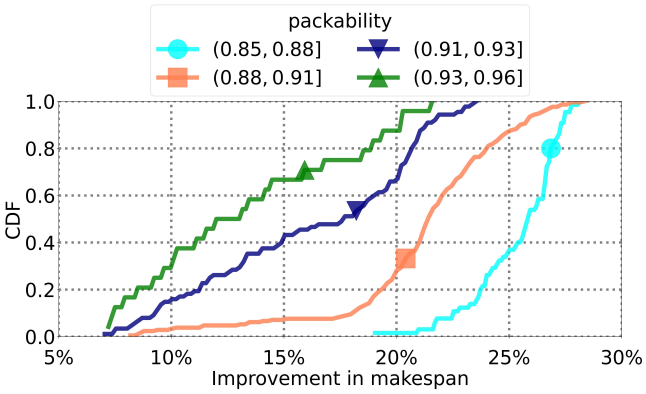

Figure 8. The impact of packability on far memory's ability to improve makespan with FAR (+17.5%) for workloads with $1.15 < m2c \leq 1.65$.

can add memory at *finer granularity*. Thus, if we do not want to invest the money required to add memory to all machines in a rack, we can reap much of the performance benefits by adding memory to a far memory server instead at a fraction of the cost. For example, for an m2c ratio in [1.75,1.79], we could achieve a median makespan improvement of 36% by adding 48 GB locally to each machine, or we could achieve 58% of that improvement by adding 30% as much far memory with FAR (+7.5%).

These results also have implications for how racks should be constructed in the future. Figure 9 demonstrates that for $m2c > 1.15$, the smallest far memory configuration FAR (+0%) achieves an average of 9.3% improved makespan relative to NOFAR (+0%). This suggests that if your workloads are even slightly memory-constrained when you build your rack, you will achieve better performance by equipping it with 39 servers and 1 memory server with an equivalent amount of memory, than by equipping it with 40 servers, despite the loss of one server's worth of cores. Furthermore, this design allows you to easily add more memory as workloads change in the future.

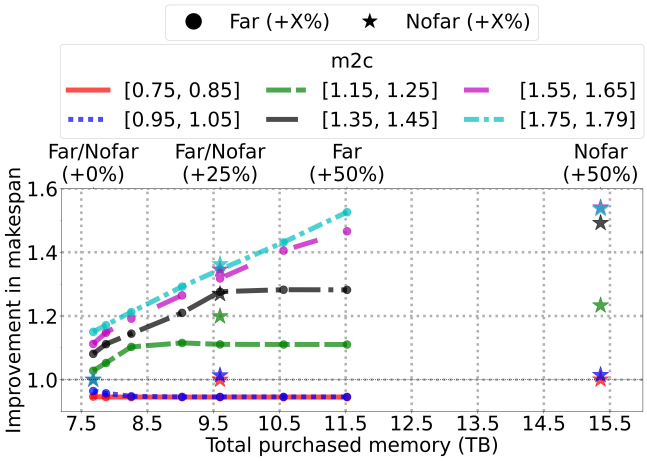

Figure 9. Makespan improvement as we add memory to the cluster in different ways, for workloads with different m2c values. Dots show addition of far memory, while stars are addition of local memory, for a given amount of purchased memory.

6.4 Microbenchmarks

We now use microbenchmarks to study CFM's performance and how it is impacted by Fastswap's design elements. The benchmarks in this subsection were executed in our testbed setup, unless noted otherwise.

6.4.1 Page Fetch Rate. In this benchmark, we measure how quickly Fastswap can fetch pages over the network. Our benchmark triggers page fetches as quickly as possible by performing memory reads that are strided by the size of a page (4 KB) such that each memory read causes a major page fault. To isolate the performance of fetches, we prevent evictions from occurring during the experiment. We run multiple instances of the benchmark process and pin each to its own core.

We evaluate Fastswap as well as a variant of Fastswap in which all page fetches (including those for the faulting page) raise interrupts on completions (Fastswap-interrupt-only). We also evaluate Infiniswap, both with and without asynchronous writes to disk enabled, and a DRAM backend, which

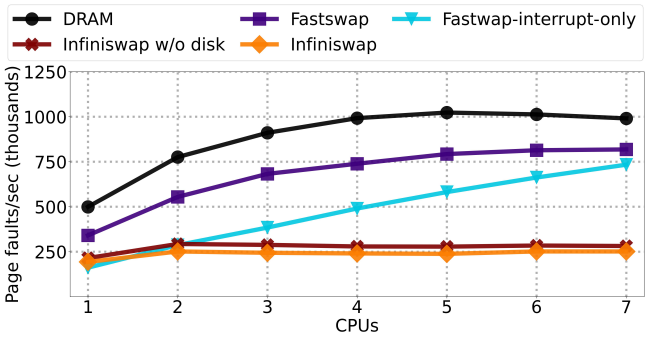

Figure 10. Page fetch rates supported by different swapping backends when eviction is disabled (prefetch set size is 8 pages).

provides an upper bound on achievable performance using Linux's swapping mechanisms.

As shown in Figure 10, the DRAM backend's fetch rate scales sublinearly and achieves a peak page fault rate of 1.02M pages/sec with 5 cores. Fastswap can achieve up to 80% of this, peaking at 818,000 pages/sec with 7 cores. Even though DRAM can copy a page in <1us [57] compared to 3.9μs[7] for an RDMA read of 4KB, Fastswap achieves 80% of the fetch rate of DRAM because of its hybrid polling-interrupt mechanism to fetch pages. DRAM has no asynchronous mechanism: it copies all pages synchronously whether for prefetching or reading the faulting page. Meanwhile, Fastswap only waits for the faulting page, and handles prefetched pages via interrupts. Fastswap-interrupt-only demonstrates the benefits of polling for the fetched page; when all pages generate interrupts, it takes even longer to finish fetching the faulting page, leading to lower fetch throughput.

Infiniswap achieves significantly lower page fault rates, peaking at 320k pages per second with 6 cores (39% of Fastswap), even without writes to disk. Unfortunately these rates are insufficient to support our experiments. For our testbed experiments described in §6.2 we observed peak page access rates of 431k pages per second with $m2c$=1.6; therefore, Infiniswap would not be able to support them.

6.4.2 Cgroup Memory Bandwidth. In this experiment, we measure the memory bandwidth that cgroups using Fastswap provide to applications. We use STREAM, a well-known industry standard benchmark [53]. This benchmark performs operations over large regions of memory, triggering both fetches and evictions. We configure the benchmark to use 4 GB of memory. As Figure 11 shows, we set the percentage of local memory to 90%, 50% and 10%. We report the Triad component of the benchmark and measure the average bandwidth instead of the peak bandwidth. Fastswap provides 6–78% higher bandwidth than Infiniswap with 1 CPU and 25–110% higher bandwidth with 4 CPUs. At 100% local memory, 1 CPU achieves 13,867 MB/s and 4 CPUs achieve 32,253 MB/s,

[7] Average 4KB RDMA read latency measured in our testbed.

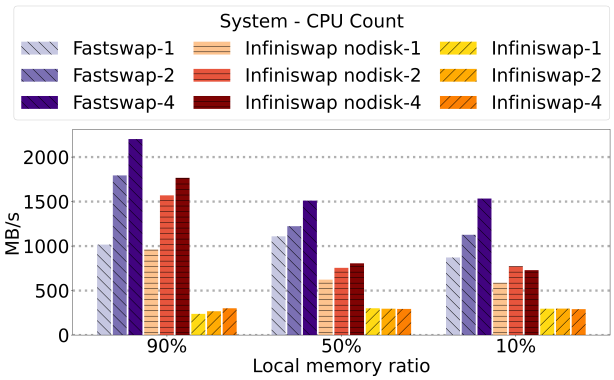

Figure 11. Memory bandwidth achieved by the STREAM benchmark for Fastswap and Infiniswap with different core counts and local memory ratios.

Workloads	% of kernel time	Kernel time reduction
linpack	61.8%	37.7%
quicksort	72.9%	30.8%
kmeans	74.3%	25.5%
tensorflow-inception	76.7%	85.3%
spark	84.9%	35.6%

Table 3. Fraction of kernel time spent on direct memory reclamation without the Fastswap reclaimer and the percentage reduction when the Fastswap reclaimer is used.

demonstrating that local memory can achieve 12.5× or 14.6× as much bandwidth as far memory, respectively. This highlights the importance of continuing research efforts on lowering swapping overheads to further improve its performance.

6.4.3 Memory Reclaimer. We evaluate the efficacy of the Fastswap reclaimer (§4.3) by measuring the percent of kernel time spent performing reclamation both *with* and *without* the reclaimer. We run each application with 50% local memory and measure the time spent doing reclamation. Table 3 shows that without the reclaimer, 61.8–84.9% of kernel time is spent performing direct reclaim. However, enabling the reclaimer reduces this significantly so that applications spend up to 85.3% less kernel time performing direct reclamation. The reason why kernel time reduction is not 100% is that the reclaimer cannot offload all memory reclaim when large memory allocations occur, *i.e.,* larger than $\alpha = 8MB$ (see §4.3).

6.4.4 Memory Rebalance Policies. We use simulation to compare the three memory `rebalance` policies we introduced before: uniform, variable, and memory-time (§5.2). We simulate the execution of 1000 workloads following our setup for §6.3 with m2c values between 1.4 to 1.8 in a cluster that mimics our testbed cluster. We use high m2c workloads to highlight the differences between memory `rebalance` policies, because this is where far memory provides the most benefit. We set the amount of far memory to 128 GB. Overall, memory-time always performs slightly better than variable, and variable

always performs significantly better than uniform. The median makespan improvement of memory-time over uniform is 12.4%, and the median improvement of memory-time over variable is 2%.

We believe memory-time performs only slightly better than variable for two reasons. First, memory-time uses the variable policy for job admission; therefore, memory-time can improve performance relative to variable only by choosing better shrink ratios. Second, memory-time's improvements over variable arise from non-linearity in job degradation profiles, but most of the jobs in this experiment have close to linear degradation profiles. This is because at these high m2c values, most jobs are memory-intensive jobs such as quicksort and memcached, which have close to linear degradation profiles, rather than jobs like spark or kmeans whose profiles are less linear (see Figure 4). We believe that a greater diversity of job degradation profiles amongst the memory-intensive applications would yield larger makespan improvements for memory-time relative to variable.

7 Related work

Hardware resource disaggregation. The idea of *disaggregating* hardware resources in datacenters has gained popularity in recent years. As a result, recent work has considered how to adapt various components of datacenters to support disaggregation, proposing new hardware designs [8, 11, 16, 25, 41, 48], operating systems [49, 58], memory abstractions [1, 59], and network stacks [21], and studying the requirements imposed on underlying networks [30]. CFM's scheduling policies and faster swapping mechanisms are complimentary to these efforts.

Far memory access. Several previous systems have used paging over a network to leverage remote memory [15, 18, 27, 29, 37, 48]. More recent efforts such as HPBD [47] and Infiniswap [34] leverage RDMA to implement swapping over the network with lower latency. Though Fastswap also implements swapping over RDMA, it overcomes several challenges that limit the latency and throughput of swapping of these existing systems (§3.2.1). Another recent approach implements "far memory" by compressing cold pages and storing them locally in DRAM [43]. With this approach, the authors were able to store about 20% of their data compressed in DRAM. However, this approach has a limited ability to address the ever-increasing demands for memory. Fabric Attached Memory [42] proposes to use far memory without paging; however, to the best of our knowledge no publicly available implementation of hardware exists yet.

Cluster scheduling. Many existing cluster schedulers such as Decima [51], Tetris [32], and others [31, 33, 38] have considered how to pack jobs onto compute clusters in order to maximize for efficient use of cluster resources such as memory, CPU, disk, and network. Cluster managers such as Borg [61], Omega [56], YARN [60], and Mesos [36] schedule jobs across machines at a large-scale (e.g., thousands of machines), while also addressing issues such as failures and heterogeneous hardware. However, none of these approaches specify how to schedule jobs when their memory can be split across local and shared remote memory; we expect that Fastswap's policies could be incorporated into many of these schedulers.

8 Conclusion

This paper studies the confluence of two trends: the increasing memory requirements of cluster workloads, and the emergence of memory disaggregation. We focus on two main questions: (1) can we develop fast swapping techniques and scheduling algorithms that make far memory feasible, and (2) can we characterize some scenarios where the use of far memory leads to reduced makespans for memory-intensive workloads. Our results suggest that the answer to both questions is "yes".

Acknowledgements

We thank our shepherd Sergey Blagodurov, the anonymous reviewers, James McCauley, and other members of NetSys for their useful feedback. We thank Peter Gao for his feedback on an early version of this work. We thank Kostadin Ilov for his technical support in the testbed cluster. Emmanuel Amaro was partially supported by a UCMEXUS-Conacyt Fellowship. This work was funded by NSF Grants 1817115, 1817116, 1704941, and was supported by Intel, VMware and Microsoft.

References

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote Regions: A Simple Abstraction for Remote Memory. In *USENIX Annual Technical Conference (USENIX ATC '18)*. 775–787.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote Memory in the Age of Fast Networks. In *ACM Symposium on Cloud Computing (SoCC'17)*. 121–127. https://doi.org/10.1145/3127479.3131612

[3] Anonymous. Accessed 2019/09/7. Frontswap. https://www.kernel.org/doc/html/latest/vm/frontswap.html.

[4] Anonymous. Accessed 2019/1/1. Recommended Intel Xeon SP Memory Configurations. http://bladesmadesimple.com/wp-content/uploads/2019/06/Intel_Xeon_SP_Memory_Recommendations_v4.pdf.

[5] Anonymous. Accessed 2019/11/3. Cgroups v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt.

[6] Anonymous. Accessed 2019/9/24. The Machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/.

[7] Anonymous. Accessed 2020/3/5. Intel Optimized LINPACK Benchmark for Linux. https://software.intel.com/en-us/mkl-linux-developer-guide-intel-optimized-linpack-benchmark-for-linux.

[8] Anonymous. Accessed 2020/3/5. Intel Rack Scale Design (Intel RSD). https://www.intel.ca/content/www/ca/en/architecture-and-technology/rack-scale-design-overview.html.

[9] Anonymous. Accessed 2020/3/5. Machine Learning in Python. https://scikit-learn.org/stable/.

[10] Anonymous. Accessed 2020/3/5. TensorFlow benchmarks. https://github.com/tensorflow/benchmarks/.

[11] Krste Asanovic. 2014. Firebox: A Hardware Building Block for 2020 Warehouse-scale Computers. In *Keynote presentation at the USENIX Conference on File and Storage Technologies (FAST'14)*.

[12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-Value Store. In *ACM SIGMETRICS Performance Evaluation Review (SIGMETRICS'12)*. 53–64. https://doi.org/10.1145/2254756.2254766

[13] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (2017), 48–54. https://doi.org/10.1145/3015146

[14] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: Designing Warehouse-scale Machines*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00874ED3V01Y201809CAC046

[15] Haogang Chen, Yingwei Luo, Xiaolin Wang, Binbin Zhang, Yifeng Sun, and Zhenlin Wang. 2008. A Transparent Remote Paging Model for Virtual Machines. In *International Workshop on Virtualization Technology*.

[16] I-Hsin Chung, Bulent Abali, and Paul Crumley. 2018. Towards a Composable Computer System. In *International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2018)*. 137–147. https://doi.org/10.1145/3149457.3149466

[17] Dan Colglazier, Joseph Jakubowski, and Jamal Ayoubi. Accessed 2019/11/1. Intel Xeon Scalable Family Balanced Memory Configurations. https://lenovopress.com/lp0742.pdf.

[18] Douglas E. Comer and James Griffioen. 1990. A New Design for Distributed Systems: The Remote Memory Model. Technical Report 90-977. Purdue University, Department of Computer Science.

[19] CCIX Consortium et al. Accessed 2019/9/24. Cache Coherent Interconnect for Accelerators (CCIX). http://www.ccixconsortium.com.

[20] GenZ Consortium et al. Accessed 2019/9/24. GenZ Consortium. https://www.genzconsortium.org.

[21] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. 2015. R2C2: A Network Stack for Rack-scale Computers. In *ACM Special Interest Group on Data Communications (SIGCOMM'15)*. 551–564. https://doi.org/10.1145/2785956.2787492

[22] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc De Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Symposium on Networked Systems Design and Implementation (NSDI'18)*. 373–387.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Symposium on Networked Systems Design and Implementation (NSDI'14)*. 401–414.

[24] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. In *European Conference on Computer Systems (EuroSys'18)*. https://doi.org/10.1145/3190508.3190524

[25] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. 2015. Beyond Processor-centric Operating Systems. In *Workshop on Hot Topics in Operating Systems (HotOS'15)*.

[26] Viacheslav Fedorov, Jinchun Kim, Mian Qin, Paul V. Gratz, and A. L. Narasimha Reddy. 2017. Speculative Paging for Future NVM Storage. In *International Symposium on Memory Systems (MEMSYS'17)*. 399–410. https://doi.org/10.1145/3132402.3132409

[27] Edward W. Felten and John Zahorjan. 1991. Issues in the Implementation of a Remote Memory Paging System. Technical Report. University of Washington, Department of Computer Science and Engineering.

[28] Werner Fischer. Accessed 2019/11/1. Optimize Memory Performance of Intel Xeon Scalable Systems. https://www.thomas-krenn.com/en/wiki/Optimize_memory_performance_of_Intel_Xeon_Scalable_systems.

[29] Michail D. Flouris and Evangelos P. Markatos. 1999. The Network RamDisk: Using Remote Memory on Heterogeneous NOWs. *Cluster Computing* 2, 4 (1999), 281–293.

[30] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *Symposium on Operating Systems Design and Implementation (OSDI'16)*. 249–264.

[31] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Symposium on Operating Systems Design and Implementation (OSDI'16)*. 99–115.

[32] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource Packing for Cluster Schedulers. In *ACM Special Interest Group on Data Communications (SIGCOMM'14)*. 455–466. https://doi.org/10.1145/2619239.2626334

[33] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. Graphene: Packing and Dependency-aware Scheduling for Data-parallel Clusters. In *Symposium on Operating Systems Design and Implementation (OSDI'16)*. 81–97.

[34] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with INFINISWAP. In *Symposium on Networked Systems Design and Implementation (NSDI'17)*. 649–667.

[35] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. 161–175. https://doi.org/10.1145/2694344.2694384

[36] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Symposium on Networked Systems Design and Implementation (NSDI'11)*. 295–308.

[37] Michael R. Hines, Mark Lewandowski, and Kartik Gopalan. 2005. Anemone: Adaptive Network Memory Engine. Technical Report TR-050128. Florida State University, Department of Computer Science.

[38] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *ACM Symposium on Operating Systems Principles (SOSP'09)*. 261–276. https://doi.org/10.1145/1629575.1629601

[39] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *USENIX Annual Technical Conference (USENIX ATC '16)*. 437–450.

[40] Uksong Kang, Hak-Soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, S. Jang, and Joo Sun Choi. 2014. Co-architecting Controllers and DRAM to Enhance DRAM Process Scaling. In *The memory forum*.

[41] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. 2016. Rack-scale Disaggregated Cloud Data Centers: The dReDBox Project Vision. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'16)*. 690–695.

[42] Kimberly Keeton. 2017. Memory-driven Computing. In *Keynote presentation at the USENIX Conference on File and Storage Technologies (FAST'17)*.

[43] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-defined Far Memory in Warehouse-scale Computers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*

(ASPLOS'19). 317–330. https://doi.org/10.1145/3297858.3304053

[44] Seok-Hee Lee. 2016. Technology Scaling Challenges and Opportunities of Memory Devices. In *IEEE International Electron Devices Meeting*.

[45] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[46] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn S. McKinley. 2016. Work Stealing for Interactive Services to Meet Target Latency. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. https://doi.org/10.1145/2851141.2851151

[47] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. 2005. Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device. In *IEEE International Conference on Cluster Computing*.

[48] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *International Symposium on Computer Architecture (ISCA'09)*. 267–278. https://doi.org/10.1145/1555754.1555789

[49] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level Implications of Disaggregated Memory. In *IEEE Symposium on High-Performance Computer Architecture (HPCA'12)*. https://doi.org/10.1109/HPCA.2012.6168955

[50] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. 2014. Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-reliability Memory. In *International Conference on Dependable Systems and Networks (DSN'14)*. 467–478. https://doi.org/10.1109/DSN.2014.50

[51] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *ACM Special Interest Group on Data Communications (SIGCOMM'19)*. 270–288. https://doi.org/10.1145/3341302.3342080

[52] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *ACM Symposium on Operating Systems Principles (SOSP'19)*. 399–413. https://doi.org/10.1145/3341301.3359657

[53] John D McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Technical Committee on Computer Architecture Newsletter* 2, 19–25 (1995).

[54] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *USENIX Annual Technical Conference (USENIX ATC'13)*. 103–114.

[55] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *ACM Symposium on Cloud Computing (SoCC'12)*. https://doi.org/10.1145/2391229.2391236

[56] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *European Conference on Computer Systems (EuroSys'13)*. 351–364. https://doi.org/10.1145/2465351.2465386

[57] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization. In *IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. 185–197. https://doi.org/10.1145/2540708.2540725

[58] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Symposium on Operating Systems Design and Implementation (OSDI'18)*. 69–87.

[59] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE Kernel RDMA Support for Datacenter Applications. In *ACM Symposium on Operating Systems Principles (SOSP'17)*. 306–324. https://doi.org/10.1145/3132747.3132762

[60] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *ACM Symposium on Cloud Computing (SOCC'13)*. https://doi.org/10.1145/2523616.2523633

[61] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In *European Conference on Computer Systems (EuroSys'15)*. https://doi.org/10.1145/2741948.2741964

[62] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2019. SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python. *arXiv e-prints* (July 2019). arXiv:1907.10121

[63] Carl A. Waldspurger, Trausti Saemundson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization Using Miniature Simulations. In *USENIX Annual Technical Conference (USENIX ATC '17)*. 487–498.

[64] Gerhard J. Woeginger. 1997. There is No Asymptotic PTAS for Two-dimensional Vector Packing. *Inform. Process. Lett.* 64, 6 (1997), 293–297.