Network Requirements for Resource Disaggregation

Peter X. Gao *Akshay Narayan* *Sagar Karandikar* *Joao Carreira* *Sangjin Han*
UC Berkeley *UC Berkeley* *UC Berkeley* *UC Berkeley* *UC Berkeley*

Rachit Agarwal *Sylvia Ratnasamy* *Scott Shenker*
Cornell University *UC Berkeley* *UC Berkeley/ICSI*

Abstract

Traditional datacenters are designed as a collection of servers, each of which tightly couples the resources required for computing tasks. Recent industry trends suggest a paradigm shift to a disaggregated datacenter (DDC) architecture containing a pool of resources, each built as a standalone resource blade and interconnected using a network fabric.

A key enabling (or blocking) factor for disaggregation will be the network – to support good application-level performance it becomes critical that the network fabric provide low latency communication even under the increased traffic load that disaggregation introduces. In this paper, we use a workload-driven approach to derive the minimum latency and bandwidth requirements that the network in disaggregated datacenters must provide to avoid degrading application-level performance and explore the feasibility of meeting these requirements with existing system designs and commodity networking technology.

1 Introduction

Existing datacenters are built using servers, each of which tightly integrates a small amount of the various resources needed for a computing task (CPU, memory, storage). While such server-centric architectures have been the mainstay for decades, recent efforts suggest a forthcoming paradigm shift towards a *disaggregated* datacenter (DDC), where each resource type is built as a standalone resource "blade" and a network fabric interconnects these resource blades. Examples of this include Facebook Disaggregated Rack [8], HP "The Machine" [13], Intel Rack Scale Architecture [19], SeaMicro [24] as well as prototypes from the computer architecture community [31, 46, 51].

These industrial and academic efforts have been driven largely by hardware architects because CPU, memory and storage technologies exhibit significantly different trends in terms of cost, performance and power scaling [10, 21, 23, 59]. This, in turn, makes it increasingly hard to adopt evolving resource technologies within a server-centric architecture (*e.g.*, the memory-capacity wall making CPU-memory co-location unsustainable [61]). By decoupling these resources, DDC makes it easier for each resource technology to evolve independently and reduces the time-to-adoption by avoiding the burdensome process of redoing integration and motherboard design.[1] In addition, disaggregation also enables fine-grained and efficient provisioning and scheduling of individual resources across jobs [40].

A key enabling (or blocking) factor for disaggregation will be the network, since disaggregating CPU from memory and disk requires that the inter-resource communication that used to be contained *within* a server must now traverse the network fabric. Thus, to support good application-level performance it becomes critical that the network fabric provide low latency communication for this increased load. It is perhaps not surprising then that prototypes from the hardware community [8, 13, 19, 24, 31, 46, 51] all rely on new high-speed network components – e.g., silicon photonic switches and links, PCIe switches and links, new interconnect fabrics, etc. The problem, however, is that these new technologies are still a long way from matching existing commodity solutions with respect to cost efficiency, manufacturing pipelines, support tools, and so forth. Hence, at first glance, disaggregation would appear to be gated on the widespread availability of new networking technologies.

But are these new technologies strictly *necessary* for disaggregation? Somewhat surprisingly, despite the many efforts towards and benefits of resource disaggregation, there has been little systematic evaluation of the network requirements for disaggregation. In this paper, we take a first stab at eval-

[1] We assume partial CPU-memory disaggregation, where each CPU has some local memory. We believe this is a reasonable intermediate step toward full CPU-memory disaggregation.

uating the *minimum* (bandwidth and latency) requirements that the network in disaggregated datacenters must provide. We define the minimum requirement for the network as that which allows us to maintain application-level performance close to server-centric architectures; i.e., at minimum, we aim for a network that keeps performance degradation small for current applications while still enabling the aforementioned qualitative benefits of resource disaggregation.

Using a combination of emulation, simulation, and implementation, we evaluate these minimum network requirements in the context of ten workloads spanning seven popular open-source systems — Hadoop, Spark, GraphLab, Timely dataflow [26, 49], Spark Streaming, memcached [20], HERD [42], and SparkSQL. We focus on current applications such as the above because, as we elaborate in §3, they represent the worst case in terms of the application *degradation* that may result from disaggregation. Our key findings are:

- Network bandwidth in the range of $40 - 100$Gbps is sufficient to maintain application-level performance within 5% of that in existing datacenters; this is easily in reach of existing switch and NIC hardware.
- Network latency in the range of $3 - 5\mu$s is needed to maintain application-level performance. This is a challenging task. Our analysis suggests that the primary latency bottleneck stems from network software rather than hardware: we find the latency introduced by the endpoint is roughly 66% of the inter-rack latency and roughly 81% of the intra-rack latency. Thus many of the switch hardware optimizations (such as terabit links) pursued today can optimize only a small fraction of the overall latency budget. Instead, work on bypassing the kernel for packet processing and NIC integration [33] could significantly impact the feasibility of resource disaggregation.
- We show that the root cause of the above bandwidth and latency requirements is the application's memory bandwidth demand.
- While most efforts focus on disaggregating at the rack scale, our results show that for some applications, disaggregation at the datacenter scale is feasible.
- Finally, our study shows that transport protocols frequently deployed in today's datacenters (TCP or DCTCP) fail to meet our target requirements for low latency communication with the DDC workloads. However, some recent research proposals [30, 36] do provide the necessary end-to-end latencies.

Taken together, our study suggests that resource disaggregation need not be gated on the availability of new networking hardware: instead, minimal performance degradation can be achieved with existing network hardware (either commodity, or available shortly).

Communication	Latency (ns)	Bandwidth (Gbps)
CPU – CPU	10	500
CPU – Memory	20	500
CPU – Disk (SSD)	10^4	5
CPU – Disk (HDD)	10^6	1

Table 1: Typical latency and peak bandwidth requirements within a traditional server. Numbers vary between hardware.

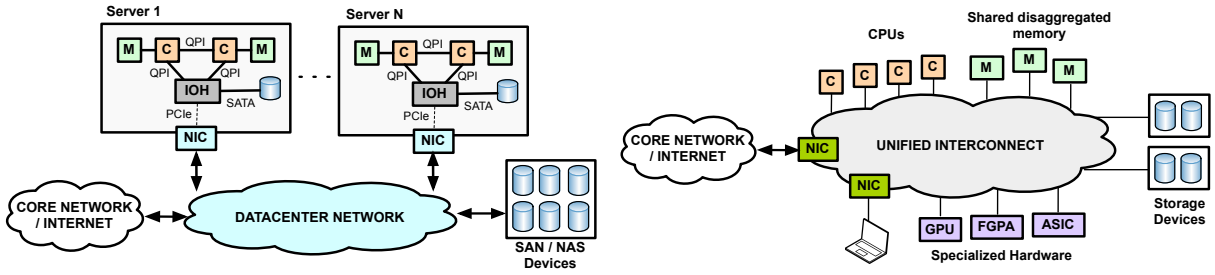
There are two important caveats to this. First, while we may not need network changes, we will need changes in hosts, for which RDMA and NIC integration (for hardware) and pFabric or pHost (for transport protocols) are promising directions. Second, our point is not that new networking technologies are not worth pursuing but that the adoption of disaggregation *need not be coupled* to the deployment of these new technologies. Instead, early efforts at disaggregation can begin with existing network technologies; system builders can incorporate the newer technologies when doing so makes sense from a performance, cost, and power standpoint.

Before continuing, we note three limitations of our work. First, our results are based on ten specific workloads spanning seven open-source systems with varying designs; we leave to future work an evaluation of whether our results generalize to other systems and workloads.[2] Second, we focus primarily on questions of network design for disaggregation, ignoring many other systems questions (*e.g.*, scheduler designs or software stack) modulo discussion on understanding latency bottlenecks. However, if the latter does turn out to be the more critical bottleneck for disaggregation, one might view our study as exploring whether the network can "get out of the way" (as often advocated [37]) even under disaggregation. Finally, our work looks ahead to an overall system that does not yet exist and hence we must make assumptions on certain fronts (e.g., hardware design and organization, data layout, etc.). We make what we believe are sensible choices, state these choices explicitly in §2, and to whatever extent possible, evaluate the sensitivity of these choices on our results. Nonetheless, our results are dependent on these choices, and more experience is needed to confirm their validity.

2 Disaggregated Datacenters

Figure 1 illustrates the high-level idea behind a disaggregated datacenter. A DDC comprises standalone hardware "blades"

[2]We encourage other researchers to extend the evaluation with our emulator. https://github.com/NetSys/disaggregation



(a) Current datacenter

(b) Disaggregated datacenter

Figure 1: High-level architectural differences between server-centric and resource-disaggregated datacenters.

for each resource type, interconnected by a network fabric. Multiple prototypes of disaggregated hardware already exist — Intel RSA [19], HP "The Machine" [13], Facebook's Disaggregated Rack [8], Huawei's DC3.0 [12], and SeaMicro [24], as well as research prototypes like FireBox [31], soNUMA [51], and memory blades [46]. Many of these systems are proprietary and/or in the early stages of development; nonetheless, in our study we draw from what information is publicly available to both borrow from and critically explore the design choices made by existing hardware prototypes.

In this section, we present our assumptions regarding the hardware (§2.1) and system (§2.2) architecture in a disaggregated datacenter. We close the section by summarizing the key open design choices that remain after our assumptions (§2.3); we treat these as design "knobs" in our evaluation.

2.1 Assumptions: Hardware Architecture

Partial CPU-memory disaggregation. In general, disaggregation suggests that each blade contains one particular resource with a direct interface to the network fabric (Fig. 1). One exception to this strict decoupling is CPU blades: each CPU blade retains some amount of *local* memory that acts as a cache for remote memory dedicated for cores on that blade[3]. Thus, CPU-memory disaggregation can be viewed as expanding the memory hierarchy to include a remote level, which all CPU blades share.

This architectural choice is reported in prior work [12, 31, 46, 47]. While we assume that partial CPU-memory disaggregation will be the norm, we go a step further and evaluate how the amount of local memory impacts *network* requirements in terms of network bandwidth and latency, and transport-layer flow completion times.

Cache coherence domain is limited to a single compute blade. As articulated by others [12, 13, 31], this has the important implication that CPU-to-CPU cache coherence traffic does not hit the network fabric. While partial CPU-memory disaggregation reduces the traffic hitting the network, cache coherence traffic can not be cached and hence directly impacts the network. This assumption is necessary because an external network fabric is unlikely to support the latency and bandwidth requirements for inter-CPU cache coherence (Table 1).

Resource Virtualization. Each resource blade must support virtualization of its resources; this is necessary for resources to be logically aggregated into higher-level abstractions such as VMs or containers. Virtualization of IO resources is widely available even today: many IO device controllers now support virtualization via PCIe, SR-IOV, or MR-IOV features [41] and the same can be leveraged to virtualize IO resources in DDC. The disaggregated memory blade prototyped by Lim et al. [46] includes a controller ASIC on each blade that implements address translation between a remote CPU's view of its address space and the addressing used internally within the blade. Other research efforts assume similar designs. We note that while the implementation of such blades may require some additional new hardware, it requires no change to existing components such as CPUs, memory modules, or storage devices themselves.

Scope of disaggregation. Existing prototypes limit the scope of disaggregation to a very small number of racks. For example, FireBox [31] envisions a single system as spanning approximately three racks and assumes that the *logical* aggregation and allocation of resources is similarly scoped; i.e., the resources allocated to a higher-level abstraction such as a VM or a container are selected from a single FireBox. Similarly, the scope of disaggregation in Intel's RSA is a single rack [19]. In contrast, in a hypothetical datacenter-scale disaggregated system, resources assigned to (for example) a single VM could be selected from anywhere in the datacenter.

Network designs. Corresponding to their assumed scope

[3]We use "remote memory" to refer to the memory located on a standalone memory blade.

Class	Application Domain	Application	System	Dataset
Class A	Off-disk Batch	WordCount	Hadoop	Wikipedia edit history [27]
	Off-disk Batch	Sort	Hadoop	Sort benchmark generator
	Graph Processing	Collaborative Filtering	GraphLab	Netflix movie rating data [22]
	Point Queries	Key-value store	Memcached	YCSB
	Streaming Queries	Stream WordCount	Spark Streaming	Wikipedia edit history [27]
Class B	In-memory Batch	WordCount	Spark	Wikipedia edit history [27]
	In-memory Batch	Sort	Spark	Sort benchmark generator
	Parallel Dataflow	Pagerank	Timely Dataflow	Friendster Social Network [9]
	In-memory Batch	SQL	Spark SQL	Big Data Benchmark [6]
	Point Queries	Key-value store	HERD	YCSB

Table 2: Applications, workloads, systems and datasets used in our study.

of disaggregation, existing prototypes assume a different network architecture for within the rack(s) that form a unit of disaggregation vs. between such racks. To our knowledge, all existing DDC prototypes use specialized – even proprietary [12, 19, 24] – network technologies and protocols within a disaggregated rack(s). For example, SeaMicro uses a proprietary Torus-based topology and routing protocol within its disaggregated system; Huawei propose a PCIe-based fabric [14]; FireBox assumes an intra-FireBox network of 1Tbps Silicon photonic links interconnected by high-radix switches [31, 43]; and Intel's RSA likewise explores the use of Silicon photonic links and switches.

Rather than simply accepting the last two design choices (rack-scale disaggregation and specialized network designs), we critically explore when and why these choices are necessary. Our rationale in this is twofold. First, these are both choices that appear to be motivated not by fundamental constraints around disaggregating memory or CPU at the hardware level, but rather by the assumption that existing networking solutions cannot meet the (bandwidth/latency) requirements that disaggregation imposes on the network. To our knowledge, however, there has been no published evaluation showing this to be the case; hence, we seek to develop quantifiable arguments that either confirm or refute the need for these choices.

Second, these choices are likely to complicate or delay the deployment of DDC. The use of a different network architecture within vs. between disaggregated islands leads to the complexity of a two-tier heterogeneous network architecture with different protocols, configuration APIs, etc., for each; e.g., in the context of their FireBox system, the authors envisage the use of special gateway devices that translate between their custom intra-FireBox protocols and TCP/IP that is used between FireBox systems; Huawei's DC3.0 makes similar assumptions. Likewise, many of the specialized technologies these systems use (e.g., Si-photonic [58]) are still far from mainstream. Hence, once again, rather than assume change is necessary, we evaluate the possibility of maintaining a uniform "flat" network architecture based on existing commodity components as advocated in prior work [28, 38, 39].

2.2 Assumptions: System Architecture

In contrast to our assumptions regarding hardware which we based on existing prototypes, we have less to guide us on the systems front. We thus make the following assumptions, which we believe are reasonable:

System abstractions for *logical* resource aggregations. In a DDC, we will need system abstractions that represent a logical aggregation of resources, in terms of which we implement resource allocation and scheduling. One such abstraction in existing datacenters is a VM: operators provision VMs to aggregate slices of hardware resources within a server, and schedulers place jobs across VMs. While not strictly necessary, we note that the VM model can still be useful in DDC.[4] For convenience, in this paper we assume that computational resources are still aggregated to form VMs (or VM-like constructs), although now the resources assigned to a VM come from distributed hardware blades. Given a VM (or VM-like) abstraction, we assign resources to VMs differently based on the *scope* of disaggregation that we assume: for rack-scale disaggregation, a VM is assigned resources from within a single rack while, for datacenter-scale disaggregation, a VM is assigned resources from anywhere in the datacenter.

Hardware organization. We assume that resources are organized in racks as in today's datacenters. We assume a "mixed" organization in which each rack hosts a mix of different types of resource blades, as opposed to a "segregated" organization in which a rack is populated with a single

[4] In particular, continuing with the abstraction of a VM would allow existing software infrastructure — i.e., hypervisors, operating systems, datacenter middleware, and applications — to be reused with little or no modification.

type of resource (e.g., all memory blades). This leads to a more uniform communication pattern which should simplify network design and also permits optimizations that aim to localize communication; e.g., co-locating a VM within a rack, which would not be possible with a segregated organization.

Page-level remote memory access. In traditional servers, the typical memory access between CPU and DRAM occurs in the unit of a cache-line size (64B in x86). In contrast, we assume that CPU blades access remote memory at the granularity of a page (4KB in x86), since page-level access has been shown to better exploit spatial locality in common memory access patterns [46]. Moreover, this requires little or no modification to the virtual memory subsystems of hypervisors or operating systems, and is completely transparent to user-level applications.

Block-level distributed data placement. We assume that applications in DDC read and write large files at the granularity of "sectors" (512B in x86). Furthermore, the disk block address space is range partitioned into "blocks", that are uniformly distributed across the disk blades. The latter is partially motivated by existing distributed file systems (*e.g.*, HDFS) and also enables better load balancing.

2.3 Design knobs

Given the above assumptions, we are left with two key system design choices that we treat as "knobs" in our study: *the amount of local memory on compute blades* and *the scope of disaggregation* (e.g., rack- or datacenter-scale). We explore how varying these knobs impacts the network requirements and traffic characteristics in DDC in the following section.

The remainder of this paper is organized as follows. We first analyze network-layer bandwidth and latency requirements in DDC (§3) *without* considering contention between network flows, then in §4 relax this constraint. We end with a discussion of the future directions in §5.

3 Network Requirements

We start by evaluating network latency and bandwidth requirements for disaggregation. We describe our evaluation methodology (§3.1), present our results (§3.2) and then discuss their implications (§3.3).

3.1 Methodology

In DDC, traffic between resources that was contained within a server is now carried on the "external" network. As with other types of interconnects, the key requirement will be low latency and high throughput to enable this disaggregation. We review the forms of communication between resources within a server in Table 1 to examine the feasibility of such a network. As mentioned in §2, CPU-to-CPU cache coherence traffic does not cross the external network. For I/O traffic to storage devices, the current latency and bandwidth requirements are such that we can expect to consolidate them into the network fabric with low performance impact, assuming we have a 40Gbps or 100Gbps network. Thus, the dominant impact to application performance will come from CPU-memory disaggregation; hence, we focus on evaluating the network bandwidth and latency required to support remote memory.

As mentioned earlier, we assume that remote memory is managed at the page granularity, in conjunction with virtual memory page replacement algorithms implemented by the hypervisor or operating system. For each paging operation there are two main sources of performance penalty: i) the software overhead for trap and page eviction and ii) the time to transfer pages over the network. Given our focus on network requirements, we only consider the latter in this paper (modulo a brief discussion on current software overheads later in this section).

Applications. We use workloads from diverse applications running on real-world and benchmark datasets, as shown in Table 2. The workloads can be classified into two classes based on their performance characteristics. We elaborate briefly on our choice to take these applications as is, rather than seek to optimize them for DDC. Our focus in this paper is on understanding whether and why networking might gate the deployment of DDC. For this, we are interested in the degradation that applications might suffer if they were to run in DDC. We thus compare the performance of an application in a server-centric architecture to its performance in the disaggregated context we consider here (with its level of bandwidth and local memory). This would be strictly worse than if we compared to the application's performance if it had been rewritten for this disaggregated context. Thus, legacy (i.e., server-centric) applications represent the worst-case in terms of potential degradation and give us a lower bound on the network requirements needed for disaggregation (it might be that rewritten applications could make do with lower bandwidths). Clearly, if new networking technologies exceed this lower bound, then all applications (legacy and "native" DDC) will benefit. Similarly, new programming models designed to exploit disaggregation can only improve the performance of all applications. The question of how to achieve improved performance through new technologies and programming models is an interesting one but beyond the scope of our effort and hence one we leave to future work.

Emulating remote memory. We run the following applications unmodified with 8 threads and reduce the amount of local memory directly accessible by the applications.

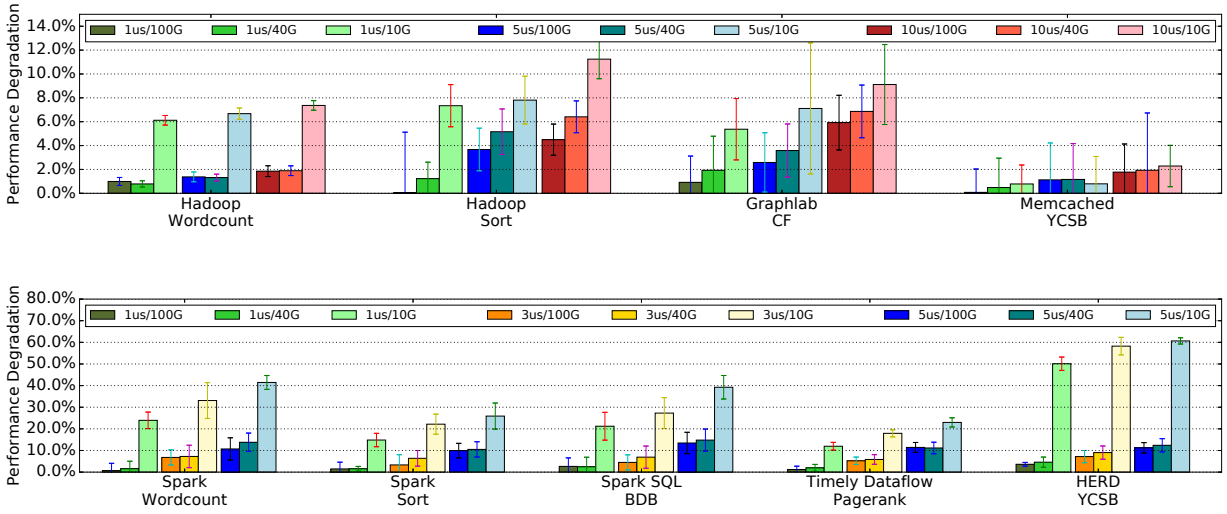


Figure 2: Comparison of application-level performance in disaggregated datacenters with respect to existing server-centric architectures for different latency/bandwidth configurations and 25% local memory on CPU blades — Class A apps (top) and Class B apps (bottom). To maintain application-level performance within reasonable performance bounds ($\sim$5% on an average), Class A apps require 5μs end-to-end latency and 40Gbps bandwidth, and Class B apps require 3μs end-to-end latency and 40$-$100Gbps bandwidth. See §3.2 for detailed discussion.

To emulate remote memory accesses, we implement a special swap device backed by the remaining physical memory rather than disk. This effectively partitions main memory into "local" and "remote" portions where existing page replacement algorithms control when and how pages are transferred between the two. We tune the amount of "remote" memory by configuring the size of the swap device; remaining memory is "local". We intercept all page faults and inject artificial delays to emulate network round-trip latency and bandwidth for each paging operation. Note that when a page fault occurs, the page is not actually swapped over the network; instead, it is swapped to the remaining part of the memory on the same machine.

We measure relative application-level performance on the basis of job completion time as compared to the zero-delay case. Thus, our results do not account for the delay introduced by software overheads for page operations and should be interpreted as *relative* performance degradations over different network configurations. Note too that the delay we inject is purely an artificial parameter and hence does not (for example) realistically model queuing delays that may result from network congestion caused by the extra traffic due to disaggregation; we consider network-wide traffic and effects such as congestion in §4.

Testbed. Each application operates on $\sim$ 125GB of data equally distributed across an Amazon EC2 cluster comprising 5 `m3.2xlarge` servers. Each of these servers has 8 vCPUs, 30GB main memory, 2×80GB SSD drives and a 1Gbps access link bandwidth. We enabled EC2's Virtual Private Network (VPC [3]) capability in our cluster to ensure no interference with other Amazon EC2 instances.

We verified that `m3.2xlarge` instances' 1Gbps access links were not a bottleneck to our experiment in two ways. First, in all cases where the network approached full utilization, CPU was fully utilized, indicating that the CPU was not blocked on network calls. Next, we ran our testbed on `c3.4xlarge` instances with 2Gbps access links (increased network bandwidth with roughly the same CPU). We verified that even with more bandwidth, all applications for which link utilization was high maintained high CPU utilization. This aligns with the conclusions drawn in [53].

We run batch applications (Spark, Hadoop, Graphlab, and Timely Dataflow) in a cluster with 5 worker nodes and 1 master node; the job request is issued from the master node. For point-query applications (memcached, HERD), requests are sent from client to server across the network. All applications are multi-threaded, with the same number of threads as cores. To compensate for the performance noise on EC2, we run each experiment 10 times and take the median result.

3.2 Results

We start by evaluating application performance in a disaggregated vs. a server-centric architecture. Figure 2 plots the performance degradation for each application under

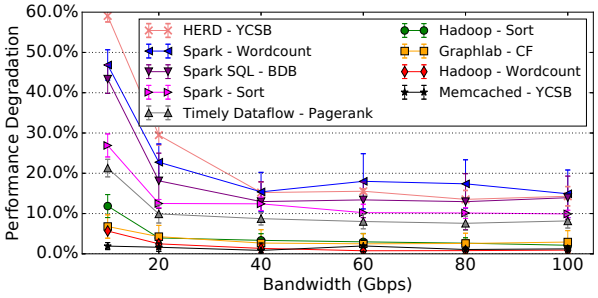


Figure 3: Impact of network bandwidth on the results of Figure 2 for end-to-end latency fixed to 5μs and local memory fixed to 25%.

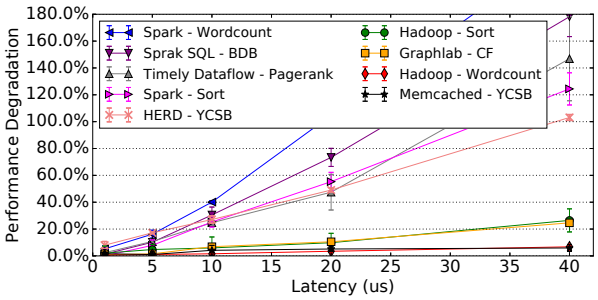


Figure 4: Impact of network latency on the results of Figure 2 for bandwidth fixed to 40Gbps and local memory fixed to 25%.

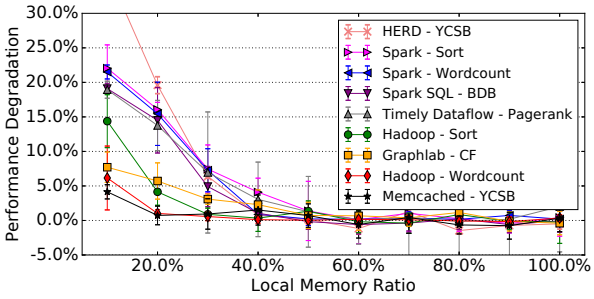


Figure 5: Impact of "local memory" on the results of Figure 2 for end-to-end latency fixed to 5μs and network bandwidth 40Gbps. Negative values are due to small variations in timings between runs.

different assumptions about the latency and bandwidth to remote memory. In these experiments, we set the local memory in the disaggregated scenario to be 25% of that in the server-centric case (we will examine our choice of 25% shortly). Note that the injected latency is constant across requests; we leave studying the effects of possibly high tail latencies to future work.

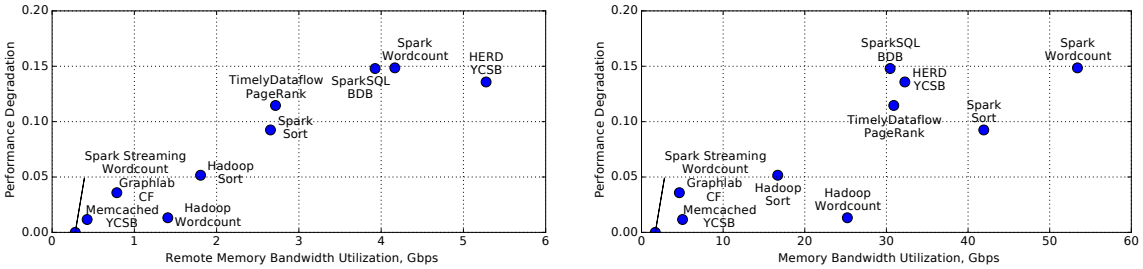
Network Provision	Class A	Class B
5μs, 40Gbps	20%	35%
3μs, 100Gbps	15%	30%

Table 3: Class B apps require slightly higher local memory than Class A apps to achieve an average performance penalty under 5% for various latency-bandwidth configurations.

From Figure 2, we see that our applications can be broadly divided into two categories based on the network latency and bandwidth needed to achieve a low performance penalty. For example, for the applications in Fig. 2 (top) — Hadoop Wordcount, Hadoop Sort, Graphlab and Memcached — a network with an end-to-end latency of 5μs and bandwidth of 40Gbps is sufficient to maintain an average performance penalty under 5%. In contrast, the applications in Fig. 2 (bottom) — Spark Wordcount, Spark Sort, Timely, SparkSQL BDB, and HERD — require network latencies of 3μs and 40 – 100Gbps bandwidth to maintain an average performance penalty under 8%. We term the former applications *Class A* and the latter *Class B* and examine the feasibility of meeting their respective requirements in §3.3. We found that Spark Streaming has a low memory utilization. As a result, its performance degradation is near zero in DDC, and we show it only in Figure 6.

Sensitivity analysis. Next, we evaluate the sensitivity of application performance to network bandwidth and latency. Fig. 3 plots the performance degradation under increasing network bandwidth assuming a fixed network latency of 5μs while Fig. 4 plots degradation under increasing latency for a fixed bandwidth of 40Gbps; in both cases, local memory is set at 25% as before. We see that beyond 40Gbps, increasing network bandwidth offers little improvement in application-level performance. In contrast, performance — particularly for Class B apps — is very sensitive to network latency; very low latencies (3 – 5μs) are needed to avoid non-trivial performance degradation.

Finally, we measure how the amount of local memory impacts application performance. Figure 5 plots the performance degradation that results as we vary the fraction of local memory from 100% (which corresponds to no CPU-memory disaggregation) down to 10%, assuming a fixed network latency and bandwidth of 5μs and 40Gbps respectively; note that the 25% values (interpolated) in Figure 5 correspond to 5μs, 40Gbps results in Figure 2. As expected, we see that Class B applications are more sensitive to the amount of local memory than Class A apps; e.g., increasing the amount of local memory from 20% to 30% roughly halves the performance degradation in Class B from approximately 15% to



(a) Swap Bandwidth Utilization

(b) Memory Bandwidth Utilization

Figure 6: Performance degradation of applications is correlated with the swap memory bandwidth and overall memory bandwidth utilization.

7%. In all cases, increasing the amount of local memory beyond 40% has little to no impact on performance degradation.

Understanding (and extrapolating from) our results. One might ask *why* we see the above requirements – i.e., what characteristic of the applications we evaluated led to the specific bandwidth and latency requirements we report? An understanding of these characteristics could also allow us to generalize our findings to other applications.

We partially answer this question using Figure 6, which plots the performance degradation of the above nine workloads against their swap and memory bandwidth[5]. Figure 6(a) and 6(b) show that an application's performance degradation is very strongly correlated with its swap bandwidth and well correlated with its memory bandwidth. The clear correlation with swap bandwidth is to be expected. That the overall memory bandwidth is also well correlated with the resultant degradation is perhaps less obvious and an encouraging result as it suggests that an application's memory bandwidth requirements might serve as a rough indicator of its expected degradation under disaggregation: this is convenient as memory bandwidth is easily measured without requiring any of our instrumentation (i.e., emulating remote memory by a special swap device, etc.). Thus it should be easy for application developers to get a rough sense of the performance degradation they might expect under disaggregation and hence the urgency of rewriting their application for disaggregated contexts.

We also note that there is room for more accurate predictors: the difference between the two figures (Figs. 6(a) and 6(b)) shows that the locality in memory access patterns does play some role in the expected degradation (since the swap bandwidth which is a better predictor captures only the subset of memory accesses that miss in local memory). Building better prediction models that account for an application's memory access pattern is an interesting question that we leave to future work.

Access Granularity. Tuning the granularity of remote memory access is an interesting area for future work. For example, soNUMA [51] accesses remote memory at cache-line size granularity, which is much smaller than page-size. This may allow point-query applications to optimize their dependence on remote memory. On the other hand, developers of applications which use large, contiguous blocks of memory may wish to use hugepages to reduce the number of page table queries and thus speed up virtual memory mapping. Since Linux currently limits (non-transparent) hugepages from being swapped out of physical memory, exploring this design option is not currently feasible.

Overall, we anticipate that programmers in DDC will face a tradeoff in optimizing their applications for disaggregation depending on its memory access patterns.

Remote SSD and NVM. Our methodology is not limited to swapping to remote memory. In fact, as long as the 3μs latency target is met, there is no limitation on the media of the remote storage. We envision that the remote memory could be replaced by SSD or forthcoming Non-Volatile Memory (NVM) technologies, and anticipate different price and performance tradeoff for these technologies.

Summary of results. In summary, supporting memory disaggregation while maintaining application-level performance within reasonable bounds imposes certain requirements on the network in terms of the end-to-end latency and bandwidth it must provide. Moreover, these requirements

[5]We use Intel's Performance Counter Monitor software [18] to read the uncore performance counters that measure the number of bytes written to and read from the integrated memory controller on each CPU. We confirmed using benchmarks designed to saturate memory bandwidth [4] that we could observe memory bandwidth utilization numbers approaching the reported theoretical maximum. As further validation, we verified that our Spark SQL measurement is consistent with prior work [54].

are closely related to the amount of local memory available to CPU blades. Table 3 summarizes these requirements for the applications we studied. We specifically investigate a few combinations of network latency, bandwidth, and the amount of local memory needed to maintain a performance degradation under 5%. We highlight these design points because they represent what we consider to be sweet spots in achievable targets both for the amount of local memory and for network requirements, as we discuss next.

3.3 Implications and Feasibility

We now examine the feasibility of meeting the requirements identified above.

Local memory. We start with the requirement of between $20-30\%$ local memory. In our experiments, this corresponds to between $1.50-2.25$GB/core. We look to existing hardware prototypes for validation of this requirement. The FireBox prototype targets 128GB of local memory shared by 100 cores leading to 1.28GB/core,[6] while the analysis in [46] uses 1.5GB/core. Further, [47] also indicates 25% local memory as a desirable setting, and HP's "The Machine" [2] uses an even larger fraction of local memory: 87%. Thus we conclude that our requirement on local memory is compatible with demonstrated hardware prototypes. Next, we examine the feasibility of meeting our targets for network bandwidth and latency.

Network bandwidth. Our bandwidth requirements are easily met: 40Gbps is available today in commodity datacenter switches *and* server NICs [16]; in fact, even 100Gbps switches and NICs are available, though not as widely [1]. Thus, ignoring the potential effects of congestion (which we consider next in §4), providing the network bandwidth needed for disaggregation should pose no problem. Moreover, this should continue to be the case in the future because the trend in link bandwidths currently exceeds that in number of cores [5, 7, 11].

Network latency. The picture is less clear with respect to latency. In what follows, we consider the various components of network latency and whether they can be accommodated in our target budget of 3μs (for Class B apps) to 5μs (for Class A apps).

Table 4 lists the six components of the end-to-end latency incurred when fetching a 4KB page using 40Gbps links, together with our estimates for each. Our estimates are based on the following common assumptions about existing datacenter networks: (1) the one-way path between servers in different racks crosses three switches (two ToR and one fabric switch) while that between servers in the same rack crosses a single ToR switch, (2) inter-rack distances of 40m and intra-rack distances of 4m with a propagation speed of 5ns/m, (3) cut-through switches.[7] With this, our round-trip latency includes the software overheads associated with moving the page to/from the NIC at both the sending and receiving endpoints (hence 2x the OS and data copy overheads), 6 switch traversals, 4 link traversals in each direction including two intra-rack and two cross-rack, and the transmission time for a 4KB page (we ignore transmission time for the page request), leading to the estimates in Table 4.

We start by observing that the network introduces three unavoidable latency overheads: (i) the data transmission time, (ii) the propagation delay; and (iii) the switching delay. Together, these components contribute to roughly 3.14μs across racks and 1.38μs within a rack.[8]

In contrast, the network software at the endpoints is a significant contributor to the end-to-end latency! Recent work reports a round-trip kernel processing time of 950 ns measured on a 2.93GHz Intel CPU running FreeBSD (see [55] for details), while [52] reports an overhead of around 1μs to copy data between memory and the NIC. With these estimates, the network software contributes roughly 3.9μs latency — this represents 55% of the end-to-end latency in our baseline inter-rack scenario and 73% in our baseline intra-rack scenario.

The end-to-end latencies we estimated in our baseline scenarios (whether inter- or intra-rack) fail to meet our target latencies for either Class B or Class A applications. Hence, we consider potential optimizations and technologies that can reduce these latencies. Two technologies show promise: RDMA and integrated NICs.

Using RDMA. RDMA effectively bypasses the packet processing in the kernel, thus eliminating the OS overheads from Table 4. Thus, using RDMA (Infiniband [15] or Omnipath [17]), we estimate a reduced end-to-end latency of 5.14μs across racks (column #4 in Table 4) and 3.38μs within a rack.

Using NIC integration. Recent industry efforts pursue the integration of NIC functions closer to the CPU [33] which would reduce the overheads associated with copying data to/from the NIC. Rosenblum *et al.* [56] estimate that such integration together with certain software optimizations can reduce copy overheads to sub-microseconds, which we estimate at 0.5μs (similar to [56]).

[6] We thank Krste Asanović for clarification on FireBox's technical specs.

[7] As before, we ignore the queuing delays that may result from congestion at switches – we will account for this in §4.

[8] Discussions with switch vendors revealed that they are approaching the fundamental limits in reducing switching delays (for electronic switches), hence we treat the switching delay as unavoidable.

Component	Baseline (μs)		With RDMA (μs)		With RDMA + NIC Integr. (μs)	
	Inter-rack	Intra-rack	Inter-rack	Intra-rack	Inter-rack	Intra-rack
OS	2×0.95	2×0.95	0	0	0	0
Data copy	2×1.00	2×1.00	2×1.00	2×1.00	2×0.50	2×0.50
Switching	6×0.24	2×0.24	6×0.24	2×0.24	6×0.24	2×0.24
Propagation (Inter-rack)	4×0.20	0	4×0.20	0	4×0.20	0
Propagation (Intra-rack)	4×0.02	4×0.02	4×0.02	4×0.02	4×0.02	4×0.02
Transmission	1×0.82	1×0.82	1×0.82	1×0.82	1×0.82	1×0.82
Total	7.04μs	5.28μs	5.14μs	3.38μs	4.14μs	2.38μs

Table 4: Achievable round-trip latency (Total) and the components that contribute to the round-trip latency (see discussion in §3.3) on a network with 40Gbps access link bandwidth (one can further reduce the **Total** by 0.5μs using 100Gbps access link bandwidth). The baseline denotes the latency achievable with existing network technology. The fractional part in each cell is the latency for one traversal of the corresponding component and the integral part is the number of traversal performed in one round-trip time (see discussion in §3.3).

Using RDMA and NIC integration. As shown in column #5 in Table 4, the use of RDMA together with NIC integration reduces the end-to-end latency to 4.14μs across racks; within a rack, this further reduces down to 2.38μs (using the same differences as in column #2 and column #3).

Takeaways. We highlight a few takeaways from our analysis:

- The overhead of network *software* is the key barrier to realizing disaggregation with current networking technologies. Technologies such as RDMA and integrated NICs that eliminate some of these overheads offer promise: reducing end-to-end latencies to 4.14μs between racks and 2.38μs within a rack. However, demonstrating such latencies in a working prototype remains an important topic for future exploration.
- Even assuming RDMA and NIC integration, the end-to-end latency across racks (4.14μs) meets our target latency only for Class A, but not Class B, applications. Our target latency for Class B apps is only met by the end-to-end latency within a rack. Thus, Class B jobs will have to be scheduled within a single rack (or nearby racks). That is, while Class A jobs can be scheduled at blades distributed across the datacenter, Class B jobs will need to be scheduled within a rack. The design and evaluation of such schedulers remains an open topic for future research.
- While new network hardware such as high-bandwidth links (e.g., 100Gbps or even 1Tbps as in [31, 43]) and high-radix switches (*e.g.*, 1000 radix switch [31]) are certainly useful, they optimize a relatively small piece of the overall latency in our baseline scenario technologies. All-optical switches also fall into this category – providing both potentially negligible switching delay and high bandwidth. That said, once we assume the benefits of RDMA and NIC integration, then the contribution of new links and switches could bring even the cross-rack latency to within our 3μs target for Class B applications, enabling true datacenter-scale disaggregation; e.g., using 100Gbps links reduces the end-to-end latency to 3.59μs between racks, extremely close to our 3μs.
- Finally, we note that managing network congestion to achieve zero or close-to-zero queuing within the network will be essential; e.g., a packet that is delayed such that it is queued behind (say) 4 packets will accumulate an additional delay of $4 \times 0.82\mu$s! Indeed, reducing such transmission delays may be the reason to adopt high-speed links. We evaluate the impact of network congestion in the following section.

4 Network Designs for Disaggregation

Our evaluation has so far ignored the impact of queuing delay on end-to-end latency and hence application performance; we remedy the omission in this section. The challenge is that queuing delay is a function of the overall network design, including: the traffic workload, network topology and routing, and the end-to-end transport protocol. Our evaluation focuses on existing proposals for transport protocols, with standard assumptions about the datacenter topology and routing. However, the input traffic workload in DDC will be very different from that in a server-centric datacenter and, to our knowledge, no models exist that characterize traffic in a DDC.

We thus start by devising a methodology that extends our experimental setup to generate an application-driven input traffic workload (§4.1), then describe how we use this traffic model to evaluate the impact of queuing delay (§4.2). Finally, we present our results on: (i) how existing transport designs perform under DDC traffic workloads (§4.3), and (ii) how existing transport designs impact end-to-end application performance (§4.4). To our knowledge, our results represent

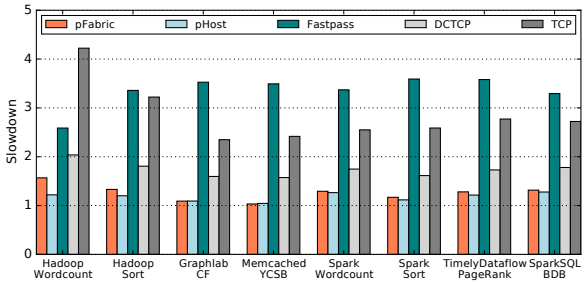


Figure 7: The performance of the five protocols for the case of 100Gbps access link capacity. The results for 40Gbps access links lead to similar conclusions. See §4.3 for discussion on these results.

the first evaluation of transport protocols for DDC.

4.1 Methodology: DDC Traffic Workloads

Using our experimental setup from §3.1, we collect a remote memory access trace from our instrumentation tool as described in §3.1, a network access trace using `tcpdump` [25], and a disk access trace using `blktrace`.

We translate the accesses from the above traces to network flows in our simulated disaggregated cluster by splitting each node into one compute, one memory, and one disk blade and assigning memory blades to virtual nodes.

All memory and disk accesses captured above are associated with a specific address in the corresponding CPU's global virtual address space. We assume this address space is uniformly partitioned across all memory and disk blades reflecting our assumption of distributed data placement (§2.2).

One subtlety remains. Consider the disk accesses at a server A in the original cluster: one might view all these disk accesses as corresponding to a flow between the compute and disk blades corresponding to A, but in reality A's CPU may have issued some of these disk accesses in response to a request from a remote server B (*e.g.*, due to a shuffle request). In the disaggregated cluster, this access should be treated as a network flow between B's compute blade and A's disk blade.

To correctly attribute accesses to the CPU that originates the request, we match network and disk traces across the cluster – e.g., matching the network traffic between B and A to the disk traffic at A – using a heuristic based on both the timestamps and volume of data transferred. If a locally captured memory or disk access request matches a local flow in our `tcpdump` traces, then it is assumed to be part of a remote read and is attributed to the remote endpoint of the network flow. Otherwise, the memory/disk access is assumed to have originated from the local CPU.

4.2 Methodology: Queuing delay

We evaluate the use of existing network designs for DDC in two steps. First, we evaluate how existing network designs fare under DDC traffic workloads. For this, we consider a suite of state-of-the-art network designs and use simulation to evaluate their network-layer performance – measured in terms of flow completion time (FCT) – under the traffic workloads we generate as above. We then return to actual execution of our applications (Table 2) and once again emulate disaggregation by injecting latencies for page misses. However, now we inject the flow completion times obtained from our best-performing network design (as opposed to the constant latencies from §3). This last step effectively "closes the loop", allowing us to evaluate the impact of disaggregation on application-level performance for realistic network designs and conditions.

Simulation Setup. We use the same simulation setup as prior work on datacenter transports [29, 30, 36]. We simulate a topology with 9 racks (with 144 total endpoints) and a full bisection bandwidth Clos topology with 36KB buffers per port; our two changes from prior work are to use 40Gbps or 100Gbps access links (as per §3), and setting propagation and switching delays as discussed in §3.3 (Table 4 with RDMA and NIC integration). We map the 5 EC2-node cluster into a disaggregated cluster with 15 blades: 5 each of compute, memory and disk. Then, we extract the flow size and inter-arrival time distribution for each endpoint pair in the 15 blades disaggregated cluster, and generate traffic using the distributions. Finally, we embed the multiple disaggregated clusters into the 144-endpoint datacenter with both rack-scale and datacenter-scale disaggreagtion, where communicating nodes are constrained to be within a rack and unconstrained, respectively.

We evaluate five protocols; in each case, we set protocol-specific parameters following the default settings but adapted to our bandwidth-delay product as recommended.

1. **TCP**, with an initial congestion window of 2.
2. **DCTCP**, which leverages ECN for enhanced performance in datacenter contexts.
3. **pFabric**, approximates shortest-job-first scheduling in a network context using switch support to prioritize flows with a smaller remaining flow size [30]. We set pFabric to have an initial congestion window of 12 packets and a retransmission timeout of 45μs.
4. **pHost**, emulates pFabric's behavior but using only scheduling at the end hosts [36] and hence allows the use of commodity switches. We set pHost to have a free token limit of 8 packets and a retransmission timeout of 9.5μs as recommended in [36].

5. **Fastpass**, introduces a centralized scheduler that schedules every packet. We implement Fastpass's scheduling algorithm in our simulator as described in [36] and optimistically assume that the scheduler's decision logic itself incurs no overhead (i.e., takes zero time) and hence we only consider the latency and bandwidth overhead of contacting the central scheduler. We set the Fastpass epoch size to be 8 packets.

4.3 Network-level performance

We evaluate the performance of our candidate transport protocols in terms of their mean slowdown [30], which is computed as follows. The slowdown for a flow is computed by dividing the flow completion time achieved in simulation by the time that the flow would take to complete if it were alone in the network. The mean slowdown is then computed by averaging the slowdown over all flows. Figure 7 plots the mean slowdown for our five candidate protocols, using 100Gbps links (all other parameters are as in §4.2).

Results. We make the following observations. First, while the relative ordering in mean slowdown for the different protocols is consistent with prior results [36], their *absolute* values are higher than reported in their original papers; e.g. pFabric and pHost both report close-to-optimal slowdowns with values close to 1.0 [30,36]. On closer examination, we found that the higher slowdowns with disaggregation are a consequence of the differences in our traffic workloads (both earlier studies used heavy-tailed traffic workloads based on measurement studies from existing datacenters). In our DDC workload, reflecting the application-driven nature of our workload, we observe many flow arrivals that appear very close in time (only observable on sub-10s of microsecond timescales), leading to high slowdowns for these flows. This effect is strongest in the case of the Wordcount application, which is why it suffers the highest slowdowns. We observed similar results in our simulation of rack-scale disaggregation (graph omitted).

4.4 Application-level performance

We now use the pFabric FCTs obtained from the above simulations as the memory access times in our emulation methodology from §3.

We measure the degradation in application performance that results from injecting remote memory access times drawn from the FCTs that pFabric achieves with 40Gbps links and with 100Gbps links, in each case considering both datacenter-wide and rack-scale disaggregation. As in §3, we measure performance degradation compared to the baseline of performance without disaggregation (i.e., injecting zero latency).

In all cases, we find that the inclusion of queuing delay

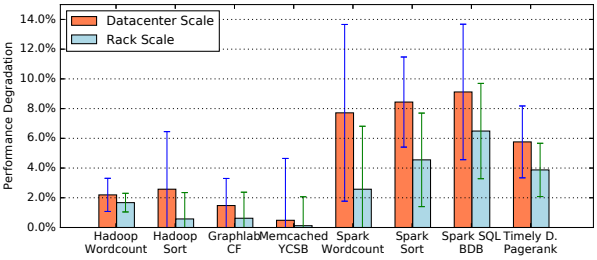


Figure 8: Application layer slowdown for each of the four applications at rack-scale and datacenter scale after injecting pFabric's FCT with 100Gbps link.

does have a non-trivial impact on performance degradation at 40 Gbps – typically increasing the performance degradation relative to the case of zero-queuing delay by between 2-3x, with an average performance degradation of 14% with datacenter-scale disaggregation and 11% with rack-scale disaggregation.

With 100Gbps links, we see (in Figure 8) that the performance degradation ranges between 1-8.5% on average with datacenter scale disaggregation, and containment to a rack lowers the degradation to between 0.4-3.5% on average. This leads us to conclude that 100Gbps links are both required and sufficient to contain the performance impact of queuing delay.

5 Future Directions

So far, we used emulation and simulation to evaluate the minimum network requirements for disaggregation. This opens two directions for future work: (1) demonstrating an end-to-end system implementation of remote memory access that meets our latency targets, and (2) investigating programming models that actively exploit disaggregation to *improve* performance. We present early results investigating the above with the intent of demonstrating the potential for realizing positive results to the above questions: each topic merits an in-depth exploration that is out of scope for this paper.

5.1 Implementing remote memory access

We previously identified an end-to-end latency target of 3-5μs for DDC that we argued could be met with RDMA. The (promising) RDMA latencies in §4 are as reported by native RDMA-based applications. We were curious about the feasibility of realizing these latencies if we were to retain our architecture from the previous section in which remote memory is accessed as a special swap device as this would provide a simple and transparent approach to utilizing remote memory.

We thus built a kernel space RDMA block device driver which serves as a swap device; i.e., the local CPU can now

Min	Avg	Median	99.5 Pcntl	Max
3394	3492	3438	4549	12254

Table 5: RDMA block device request latency(ns)

swap to remote memory instead of disk. We implemented the block device driver on a machine with a 3 GHz CPU and a Mellanox 4xFDR Infiniband card providing 56 Gbps bandwidth. We test the block device throughput using `dd` with direct IO, and measure the request latency by instrumenting the driver code. The end-to-end latency of our approach includes the RDMA request latency and the latency introduced by the kernel swap itself. We focus on each in turn.

RDMA request latency. A few optimizations were necessary to improve RDMA performance in our context. First, we *batch* block requests sent to the RDMA NIC and the driver waits for all the requests to return before notifying the upper layer: this gave a block device throughput of only 0.8GB/s and latency around 4-16us. Next, we *merge* requests with contiguous addresses into a single large request: this improved throughput to 2.6GB/s (a 3*x* improvement). Finally, we allow *asynchronous* RDMA requests: we created a data structure to keep track of outgoing requests and notify the upper layer immediately for each completed request; this improves throughput to 3.3GB/s which is as high as a local RamFS, and reduces the request latency to 3-4us (Table 5). This latency is within 2x of latencies reported by native RDMA applications which we view as encouraging given the simplicity of the design and that additional optimizations are likely possible.

Swap latency. We calculated the software overhead of swapping on a commodity desktop running Linux 3.13 by simultaneously measuring the times spent in the page fault handler and accessing disk. We found that convenient measurement tools such as `ftrace` and `printk` introduce unacceptable overhead for our purposes. Thus, we wrap both the body of the `__do_page_fault` function and the call to the `swapin_readahead` function (which performs a swap from disk) in `ktime_get` calls. We then pack the result of the measurement for the `swapin_readahead` function into the unused upper 16-bits of the return value of its caller, `do_swap_page`, which propagates the value up to `__do_page_fault`.

Once we have measured the body of `__do_page_fault`, we record both the latency of the whole `__do_page_fault` routine (25.47μs), as well as the time spent in `swapin_readahead` (23.01μs). We subtract these and average to find that the software overhead of swapping is 2.46μs. This number is a lower-bound on the software overhead of the handler, because we assume that all of `swapin_readahead` is a "disk access".

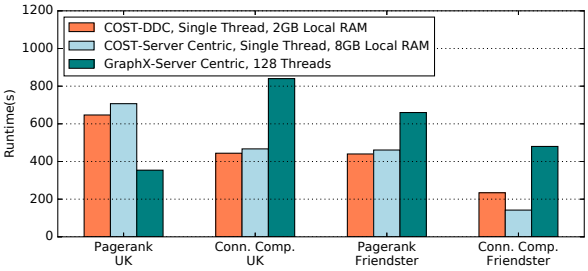


Figure 9: Running COST in a simulated DDC. COST-DDC is 1.48 to 2.05 faster than GraphX-Server Centric except for one case. We use two datasets in our evaluation, UK-2007-05 (105m nodes, 3.7b edges), and Friendster (65m nodes, 1.8b edges)

In combination with the above RDMA latencies, these early numbers suggest that a simple system design for low-latency access to remote memory could be realized.

5.2 Improved performance via disaggregation

In the longer term, one might expect to re-architect applications to actively exploit disaggregation for improved performance. One promising direction is for applications to exploit the availability of low-latency access to large pools of remote memory [46]. One approach to doing so is based on extending the line of argument in the COST work [48] by using remote memory to avoid parallelization overheads. COST is a single machine graph engine that outperforms distributed graph engines like GraphX when the graph fits into main memory. The RDMA swap device enables COST to use "infinite" remote memory when the graph is too large. We estimate the potential benefits of this approach with the following experiment. First, to model an application running in a DDC, we set up a virtual machine with 4 cores, 2GB of local memory, and access to an "infinitely" large remote memory pool by swapping to an RDMA-backed block device. Next, we consider two scenarios that represent server-centric architecture. One is a server with 4 cores and 8GB of local memory (25% larger than the DDC case as in previous sections) and an "infinitely" large local SSD swap – this represents the COST baseline in a server-centric context. Second, we evaluate GraphX using a 16-node `m2.4xlarge` cluster on EC2 – this represents the scale-out approach in current server-centric architecture. We run Pagerank and Connected Components using COST, a single-thread graph compute engine over three large graph datasets. COST `mmaps` the input file, so we store the input files on another RDMA-backed block device. Figure 9 shows the application

runtime of COST-DDC, COST-SSD and GraphX-Server Centric. In all but one case, COST-DDC is 1.48 to 2.05 times faster than the GraphX (server-centric) scenario and slightly better than the server-centric COST scenario (the improvement over the latter grows with increasing data set size). Performance is worse for Pagerank on the UK-2007-5 dataset, consistent with the results in [48] because the graph in this case is more easily partitioned.

Finally, another promising direction for improving performance is through better resource utilization. As argued in [40, 46], CPU-to-memory utilization for tasks in today's datacenters varies by three orders of magnitude across tasks; by "bin packing" on a much larger scale, DDC should achieve more efficient statistical multiplexing, and hence higher resource utilization and improved job completion times. We leave an exploration of this direction to future work.

6 Related Work and Discussion

As mentioned earlier, there are many recent and ongoing efforts to prototype disaggregated hardware. We discussed the salient features of these efforts inline throughout this paper and hence we only briefly elaborate on them here.

Lim et al. [46, 47] discuss the trend of growing peak compute-to-memory ratio, warning of the "memory capacity wall" and prototype a disaggregated memory blade. Their results demonstrate that memory disaggregation is feasible and can even provide a 10x performance improvement in memory constrained environments.

Sudan et al. [57] use an ASIC based interconnect fabric to build a virtualized I/O system for better resource sharing. However, these interconnects are designed for their specific context; the authors neither discuss network support for disaggregation more broadly nor consider the possibility of leveraging known datacenter network technologies to enable disaggregation.

FireBox [31] proposes a holistic architecture redesign of datacenter racks to include 1Tbps silicon photonic links, high-radix switches, remote nonvolatile memory, and System-on-Chips (SoCs). Theia [60] proposes a new network topology that interconnects SoCs at high density. Huawei's DC3.0 (NUWA) system uses a proprietary PCIe-based interconnect. R2C2 [34] proposes new topologies, routing and congestion control designs for rack-scale disaggregation. None of these efforts evaluate network requirements based on existing workloads as we do, nor do they evaluate the effectiveness of existing network designs in supporting disaggregation or the possibility of disaggregating at scale.

In an early position paper, Han et al. [40] measure – as we do – the impact of remote memory access latency on application-level performance within a single machine. Our work extends this understanding to a larger set of workloads and concludes with more stringent requirements on latency and bandwidth than Han et al. do, due to our consideration of Class B applications. In addition, we use simulation and emulation to study the impact of queueing delay and transport designs which further raises the bar on our target network performance.

Multiple recent efforts [35, 42, 45, 52] aim to reduce the latency in networked applications through techniques that bypass the kernel networking stack, and so forth. Similarly, efforts toward NIC integration by CPU architects [33] promise to enable even further latency-saving optimizations. As we note in §3.3, such efforts are crucial enablers in meeting our latency targets.

Distributed Shared Memory (DSM) [32, 44, 50] systems create a shared address space and allow remote memory to be accessed among different endpoints. While this is a simple programming abstraction, DSM incurs high synchronization overhead. Our work simplifies the design by using remote memory only for paging, which removes synchronization between the endpoints.

Based on our knowledge of existing designs and prototypes [12, 13, 31, 46, 47], we assume partial memory disaggregation and limit the cache coherence domain to one CPU. However, future designs may relax these assumptions, causing more remote memory access traffic and cache coherence traffic. In these designs, specialized network hardware may become necessary.

7 Conclusion

This paper is a preliminary study; we have identified numerous directions for future work before disaggregation is deployable. Most important among these are the adoption of low-latency network software and hardware at endpoints, the design and implementation of a "disaggregation-aware" scheduler, and the creation of new programming models which exploit a disaggregated architecture. We believe that quantified, workload-driven studies such as that presented in this paper can serve to inform these ongoing and future efforts to build DDC systems.

Acknowledgements

We thank our shepherd Orran Krieger and the anonymous reviewers for their excellent feedback. We thank Krste Asanović for clarification on FireBox's technical specs. We thank Kostadin Ilov for his technical support on our experiments. This work is supported by Intel, NSF Grant 1420064 and Grant 1216073.

References

[1] 100G CLR4 White Paper. http://www.intel.com/content/www/us/en/research/intel-labs-clr4-white-paper.html.

[2] A look at The Machine. https://lwn.net/Articles/655437/.

[3] Amazon VPC. https://aws.amazon.com/vpc/.

[4] Bandwidth: a memory bandwidth benchmark. http://zsmith.co/bandwidth.html.

[5] Bandwidth Growth and The Next Speed of Ethernet. http://goo.gl/C5lovt.

[6] Berkeley Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[7] Big Data System research: Trends and Challenges. http://goo.gl/38qr1O.

[8] Facebook Disaggregated Rack. http://goo.gl/6h2Ut.

[9] Friendster Social Network. https://snap.stanford.edu/data/com-Friendster.html.

[10] Graphics Processing Unit. http://www.nvidia.com/object/what-is-gpu-computing.html.

[11] Here's How Many Cores Intel Corporation's Future 14-Nanometer Server Processors Will Have. http://goo.gl/y2nWOR.

[12] High Throughput Computing Data Center Architecture. http://www.huawei.com/ilink/en/download/HW_349607.

[13] HP The Machine. http://www.hpl.hp.com/research/systems-research/themachine/.

[14] Huawei NUWA. http://nuwabox.com.

[15] InfiniBand. http://www.infinibandta.org/content/pages.php?pg=about_us_infiniband.

[16] Intel Ethernet Converged Network Adapter XL710 10/40 GbE. http://www.intel.com/content/www/us/en/network-adapters/converged-network-adapters/ethernet-xl710-brief.html.

[17] Intel Omnipath. http://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html.

[18] Intel Performance Counter Monitor. https://software.intel.com/en-us/articles/intel-performance-counter-monitor.

[19] Intel RSA. http://www.intel.com/content/www/us/en/architecture-and-technology/rsa-demo-x264.html.

[20] Memcached - A Distributed Memory Object Caching System. http://memcached.org.

[21] Memristor. http://www.memristor.org/reference/research/13/what-are-memristors.

[22] Netflix Rating Trace. http://www.select.cs.cmu.edu/code/graphlab/datasets/.

[23] Non-Volatile Random Access Memory. https://en.wikipedia.org/wiki/Non-volatile_random-access_memory.

[24] SeaMicro Technology Overview. http://seamicro.com/sites/default/files/SM_TO01_64_v2.5.pdf.

[25] "tcpdump". http://www.tcpdump.org.

[26] Timely Dataflow. https://github.com/frankmcsherry/timely-dataflow.

[27] Wikipedia Dump. https://dumps.wikimedia.org/.

[28] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. SIGCOMM 2008.

[29] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). SIGCOMM 2010.

[30] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. SIGCOMM 2013.

[31] K. Asanović. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. FAST 2014.

[32] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. PPOPP 1990.

[33] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt. Integrated Network Interfaces for High-bandwidth TCP/IP. ASPLOS 2006.

[34] P. Costa, H. Ballani, K. Razavi, and I. Kash. R2C2: A Network Stack for Rack-scale Computers. SIGCOMM 2015.

[35] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. NSDI 2014.

[36] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed Near-optimal Datacenter Transport Over Commodity Network Fabric. CoNEXT 2015.

[37] A. Greenberg. SDN for the Cloud. SIGCOMM 2015.

[38] A. Greenberg, J. Hamilton, D. Maltz, and P. Patel. The Cost of a Cloud: Research Problems in Data Center Networks. ACM SIGCOMM CCR 2009.

[39] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. SIGCOMM 2009.

[40] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network Support for Resource Disaggregation in Next-generation Datacenters. HotNets 2013.

[41] Intel LAN Access Division. An Introduction to SR-IOV Technology. `http://goo.gl/m7jP3`.

[42] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. SIGCOMM 2014.

[43] S. Kumar. Petabit Switch Fabric Design. Master's thesis, EECS Department, University of California, Berkeley, 2015.

[44] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. TOCS 1989.

[45] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-Value Storage. NSDI 2014.

[46] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. ISCA 2009.

[47] K. Lim, Y. Turner, J. R. Santos, A. Auyoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level Implications of Disaggregated Memory. HPCA 2012.

[48] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at What Cost? HotOS 2015.

[49] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. SOSP 2013.

[50] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant Software Distributed Shared Memory. USENIX ATC 2015.

[51] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. ASPLOS 2014.

[52] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. TOCS 2015.

[53] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. NSDI 2015.

[54] P. S. Rao and G. Porter. Is Memory Disaggregation Feasible?: A Case Study with Spark SQL. ANCS 2016.

[55] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. USENIX ATC 2012.

[56] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. Its Time for Low Latency. HotOS 2011.

[57] K. Sudan, S. Balakrishnan, S. Lie, M. Xu, D. Mallick, G. Lauterbach, and R. Balasubramonian. A Novel System Architecture for Web Scale Applications Using Lightweight CPUs and Virtualized I/O. HPCA 2013.

[58] C. Sun, M. T. Wade, Y. Lee, J. S. Orcutt, L. Alloatti, M. S. Georgas, A. S. Waterman, J. M. Shainline, R. R. Avizienis, S. Lin, et al. Single-chip Microprocessor that Communicates Directly Using Light. *Nature 2015*.

[59] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. RAID 2009.

[60] M. Walraed-Sullivan, J. Padhye, and D. A. Maltz. Theia: Simple and Cheap Networking for Ultra-Dense Data Centers. HotNets-XIII.

[61] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News, March 1995*.