Clio: A Hardware-Software Co-Designed Disaggregated Memory System

Zhiyuan Guo*, Yizhou Shan*, Xuhao Luo, Yutong Huang, Yiying Zhang
University of California, San Diego

Abstract

Memory disaggregation has attracted great attention recently because of its benefits in efficient memory utilization and ease of management. Research on memory disaggregation so far has taken a software approach, running disaggregated memory management software either at servers that act as disaggregated memory nodes or at servers on the client side. This paper proposes a hardware-based disaggregated memory device, *Clio*, that manages disaggregated memory at the device side with novel software-hardware co-designs. Clio includes a hardware-based virtual memory system, a customized network stack, and a framework for computation offloading. Clio achieves low median and tail latency, high throughput, excellent scalability, and low energy cost.

1 Introduction

Modern data-center applications like graph computing, data analytics, and deep learning have increasing demand for access to large amounts of memory [5]. Unfortunately, servers are facing *memory capacity walls* because of pin, space, and power limitations [28, 32, 71]. Going forward, it is imperative for datacenters to seek solutions that can go beyond what a (local) machine can offer, *i.e.*, using remote memory. At the same time, data centers are seeing the needs from management and resource utilization perspectives to *disaggregate* resources [13, 63, 68]—separating hardware resources into different network-attached pools that can be scaled and managed independently. These real needs have pushed the idea of memory disaggregation (*MemDisagg* for short): organizing computation and memory resources as two separate network-attached pools, one with compute nodes (*CN*s) and one with memory nodes (*MN*s).

So far, MemDisagg research has all taken one of two approaches, building/emulating MNs with either regular servers [5, 25, 48, 55, 56] or raw memory devices with no processing power [27, 39, 40, 65]. The fundamental issues of server-based approaches such as RDMA-based systems are the monetary cost of a host server and the inherent performance and scalability limitations caused by the way NICs interact with the host server's virtual memory system. Raw-device-based solutions have low costs. However, they introduce performance, security, and management problems because when MNs have no processing power, all the data and control planes have to be handled at CNs [65].

Server-based MNs and MNs with no processing power are two extreme approaches of building MNs. This work seeks a sweet spot in the middle by proposing a hardware-based MemDisagg solution that has the right amount of processing power at MNs. Furthermore, we take a clean-slate approach by starting from the requirements of MemDisagg and designing a MemDisagg-native system.

We built an open-source, distributed programmable hardware-based disaggregated memory framework. With this platform, applications running at different CNs can allocate and access memory from multiple MNs using a unified virtual memory interface (*e.g.*, allocate a remote virtual memory region and read/write to it at byte granularity), and they can offload certain computation to MNs. This paper presents how we built a basic, single-MN hardware platform (called *Clio*) in this framework, leaving questions like how to distribute/migrate memory across MNs and how to handle MN failures to a future paper.

Clio includes a CN-side user-space library called *CLib* and a new hardware-based MN device called *CBoard*. Multiple application processes running on different CNs could allocate memory from the same CBoard, with each process having its own remote virtual memory address space. A CBoard consists of two main components: 1) a hardware chip that integrates a thin network stack and a virtual memory system to handle data requests (the *fast path*), and 2) an ARM processor that runs software to handle metadata requests and assists the fast path with background tasks (the *slow path*).

In building Clio, we explore new requirements, challenges, and benefits of MemDisagg. Specifically, we answer three important research questions.

First, **how does the design and implementation of a dedicated hardware MN differ from server and programmable NIC designs?** Current MemDisagg solutions rely on a host server (its OS and MMU) to provide a virtual memory system so that accesses to the memory are protected and flexible. Using a whole server just for the virtual memory system is overkill and unnecessarily adds monetary and energy costs to MemDisagg. Another possibility is to

*Both authors contributed equally.

use a low-power processor (*e.g.*, ARM) in a SmartNIC to run the virtual memory system [41]. However, doing so has high performance impact mainly because the virtual memory system is on a separate chip from the NIC. Overall, server-based approaches have cost overheads while SmartNIC solutions have performance overheads. We took a clean-slate approach by building a hardware-based virtual memory system that is integrated with a customized hardware network stack, both of which are designed specifically for handling virtual memory requests sent over the network.

Second, **how can a low-cost MN host TBs of memory and support thousands of concurrent application processes?** Different from traditional (local) memory, an MN is intended to be shared by many applications running at different CNs, and the more applications it can support, the more efficiently its memory can be utilized. Thus, we aim to have each MN host TBs of memory for thousands of concurrent applications processes. However, a hardware design is constrained by the limited resources in a hardware chip such as on-chip memory. Compared to traditional software-based virtual memory systems, how can an MN use orders of magnitude less resources while achieving orders of magnitude higher scalability? Current solutions like RDMA NIC resort to caching, by swapping metadata between on-chip memory and host server memory, which inevitably comes with performance overhead (as high as 4× for a cache miss [64]).

Our clean-slate approach is to carefully examine each virtual-memory and networking task and to redesign them to 1) eliminate states and metadata whenever possible (*e.g.*, by minimizing indirection), 2) move complex but non-performance-critical states, metadata, and tasks to the software slow path, 3) shift functionalities to the CN (CLib) to reduce MN's complexity (*e.g.*, our network transport runs at CLib, and MN is "transport-less"), and 4) design bounded-size, inherently scalable data structures. As a result, each MN (CBoard) could support TBs of memory and thousands of application processes with only 1.5 MB on-chip memory.

Third, **how to minimize tail latency in a MemDisagg system**? Tail latency is important in data centers especially for workloads that have large fanouts (*e.g.*, Spark jobs). Although much effort has focused on improving the network and core scheduling for low tail latency [16, 30, 33, 51, 54], the memory system has largely been overlooked. However, the (remote) memory system is what contributes to extreme long tails in a MemDisagg system. For example, RDMA's round-trip latency is around 1–2 μs in the common case, but its tail could be as long as 16.8 ms (Figure 6 and §2.2).

We reexamine traditional memory system and propose a set of novel mechanisms to bound Clio's tail latency. Our core idea is to include *all* the functionalities that are needed to fulfill all types of data requests in one hardware pipeline and to make this hardware pipeline *performance deterministic*. This pipeline takes one incoming data unit every cycle (*i.e.*, no pipeline stalls) and completes every request in a fixed number of cycles, which yields 100 Gbps throughput, 2.5 μs at median and 3.2 μs at 99-percentile end-to-end latency (Figure 7). Two major technical hurdles in achieving this performance are to perform page table lookups and to handle page faults in a bounded, short time period. For the former, we propose a new *overflow-free* hash-based page table that bounds all page table lookups to *at most one DRAM access* (instead of the long page table walk in a traditional CPU architecture). For the latter, we propose a new mechanism to handle page faults in hardware with bounded cycles (instead of the costly process of interrupt and handling page faults in the OS).

We prototyped CBoard with an FPGA and built three applications using Clio: a FaaS-style image compression utility, a radix-tree index, and a key-value store. We compared Clio with native RDMA, two RDMA-based disaggregated/remote memory systems [34, 65], a software emulation of hardware-based disaggregated memory [56], and a software-based SmartNIC [43]. Clio scales much better and has orders of magnitude lower tail latency than RDMA, while achieving similar throughput and median latency as RDMA (even with the slower FPGA frequency in our prototype). Clio has 1.1× to 3.4× energy saving compared to CPU-based and SmartNIC-based disaggregated memory systems and is 2.7× faster than SmartNIC solutions.

2 Memory Disaggregation

Resource disaggregation separates different types of resources into different pools, each of which can be independently managed and scaled. Applications can allocate resources from any node in a resource pool, resulting in tight resource packing. Because of these benefits, many datacenters have adopted the idea of disaggregation, often at the storage layer [4, 6, 7, 13, 20, 62, 68]. With the success of disaggregated storage, researchers in academia and industry have also sought ways to disaggregate memory (and persistent memory) [5, 11, 25, 31, 39, 40, 49, 52, 55, 56, 57, 65, 69]. Different from storage disaggregation, MemDisagg needs to achieve at least an order of magnitude higher performance and it should offer a byte-addressable interface. Thus, MemDisagg poses new challenges and requires new designs. This section discusses the requirements of MemDisagg and why existing solutions cannot fully meet them.

2.1 MemDisagg Design Goals

In general, MemDisagg has the following features, some of which are hard requirements while others are desired goals.

R1: Hosting large amounts of memory. To keep the number of memory devices and total cost of a cluster low, each MN should host hundreds GBs to few TBs of memory that is expected to be close to fully utilized. Furthermore, we should allow applications to create and use many disjoint memory regions to make user programming flexible.

R2: Supporting huge number of concurrent clients. To

ensure tight and efficient resource packing, we should allow many (*e.g.*, thousands of) client processes running on tens of CNs to access and share an MN. This scenario is especially important for new data-center trends like serverless computing and microservices.

R3: Low-latency and high-throughput. We envision future systems to have a new memory hierarchy, where disaggregated memory is larger and slower than local memory but still faster than storage. We believe a good performance target of MemDisagg is to match the state-of-the-art network speed, *i.e.*, 100 Gbps throughput (for bigger requests) and sub-2 μs median end-to-end latency (for smaller requests).

R4: Low tail latency. Maintaining a low tail latency is important in meeting SLOs in data centers. Long tails like RDMA's 16.8 ms remote memory access can be detrimental to applications that are short running (*e.g.*, serverless computing workloads) or have large fan-outs or big DAGs (because they need to wait for the slowest step to finish) [17].

R5: Protected memory accesses. As an MN can be shared by multi-tenant applications running at CNs, we should properly isolate memory spaces used by them. Moreover, to prevent compromised or malicious system software running at CNs from reading/writing arbitrary memory at MNs, CNs should not directly access MNs' physical memory and MNs should check the permission of memory accesses.

R6: Low cost. A major goal and benefit of resource disaggregation is cost reduction. Previous work [56] has shown that MemDisagg could improve memory resource utilization by around 50% (*i.e.*, a MemDisagg cluster only needs to host half of the memory compared to a non-disaggregated cluster). This means that 1) a MemDisagg system should aim to have close-to-full utilization of its memory and have minimal memory waste, and 2) building and running an MN should not double the cost of hosting memory, as such a MemDisagg system would cost even more than no disaggregation. Using a server to build an MN is thus not a good option, since a server box costs more than the DRAM it hosts.

R7: Flexible. With the fast development of datacenter applications, hardware, and network, a sustainable MemDisagg solution should be flexible and extendable, for example, to support high-level APIs like pointer chasing [3, 55], to offload some application logic to memory devices [55, 58], or to incorporate different network transports [9, 26, 46] and congestion control algorithms [36, 38, 59].

2.2 Server-Based Disaggregated Memory

MemDisagg research so far has mainly taken a server-based approach by using regular servers as MNs [5, 25, 55, 56, 69], usually on top of RDMA. The common limitation of these systems is their reliance on a host server, violating **R6**.

RDMA has a set of scalability and tail-latency problems. A process (P_M) running at an MN needs to allocate memory in its virtual memory address space and *register* the allocated memory (called a memory region, or MR) with the RDMA NIC (RNIC). The host OS and MMU set up and manage the page table that maps P_M's virtual addresses (*VA*s) to physical memory addresses (*PA*s). To avoid always accessing host memory for address mapping, RNICs cache page table entries (PTEs), but when more PTEs are accessed than what this cache can hold, RDMA performance degrades significantly (Figure 5 and [19, 66]). Similarly, RNICs cache MR metadata and incur degraded performance when the cache fills. Thus, RDMA has serious performance issues with either large memory (PTEs) or many disjoint memory regions (MRs), violating **R1**. Moreover, RDMA uses a slow way to support on-demand allocation: the RNIC interrupts the host OS for handling page faults. From our experiments, a faulting RDMA access is 14100× slower than a no-fault access. Page faults happen at the initial accesses to allocated virtual memory addresses, causing long tails (violating **R4**).

To mitigate the above performance and scalability issues, most RDMA-based systems today [19, 66] preallocate a big MR with huge pages and pin it in physical memory, which results in inefficient memory space utilization and violates **R6**. Even with this approach, there can still be a scalability issue (**R2**), as RDMA uses MR as the protection domain and needs to create one MR for each client that needs isolation.

In addition to problems caused by RDMA's memory system design, reliable RDMA, the mode used by most MemDisagg solutions, suffers from a connection (QP) scalability issue, also violating **R2**. Finally, today's RNICs violate **R7** because of their rigid one-sided RDMA interface and the close-sourced, hardware-based transport implementation. Solutions like 1RMA [59], Swift [36], HPCC [38], IRN [45], and Tonic [9] mitigate the above issues by either onloading part of the transport back to software or proposing new hardware design.

LegoOS [56] is a distributed operating system designed for resource disaggregation. Its MN includes a virtual memory system that maps VAs of application processes running at CNs directly to MN PAs. Clio's MN performs the same type of address translation. However, LegoOS emulates MN devices using regular servers and builds its virtual memory system in software, which has a stark difference from a hardware-based virtual memory system. For example, LegoOS uses a thread pool that handles incoming memory requests by looking up a hash table for address translation and permission checking. This software approach is the major performance bottleneck in LegoOS (§7), violating **R3**. Moreover, LegoOS uses RDMA for its network communication hence inherits its limitations.

2.3 Physical Disaggregated Memory

One way to build MemDisagg without a host server is to treat it as raw, physical memory, a model we call *PDM*. The PDM model has been adopted by a set of coherent interconnect proposals [15, 24], HPE's Memory-Driven Computing project [21, 27, 29, 67], and a recent research project that

emulates PDM with regular servers [65]. To prevent applications from accessing raw physical memory, these solutions add an indirection layer at CNs in hardware [15, 24] or software [65] to map client process VAs or keys to MN PAs.

There are several common problems with all the PDM solutions. First, because MNs in PDM are raw memory, CNs need multiple network round trips to access an MN for complex operations like pointer chasing and concurrent operations that need synchronization [65], violating **R3** and **R7**. Second, PDM requires the client side to manage disaggregated memory, which is much harder and performs worse compared to memory-side management (violating **R3**). For example, CNs need to coordinate with each other or use a global server [65] to perform tasks like memory allocation. Third, exposing physical memory creates potential security issues (**R5**). MNs have to trust that CNs will never access beyond their allocated physical memory regions. Finally, all existing PDM solutions require physical memory pinning at MNs, causing memory wastes and violating **R6**.

In addition to the above problems, none of the coherent interconnects or HPE's Memory-Driven Computing have been fully built. When they do, they will require new hardware at all endpoints and new switches. Moreover, the interconnects automatically make caches at different endpoints coherent, which could affect performance **R3** (because of network communication) and not always necessary.

3 Clio Overview

Clio co-designs software with hardware, CNs with MNs, and network stack with virtual memory system, so that at the MN, the entire data path is handled in hardware with high throughput, low (tail) latency, and minimal hardware resources. This section gives an overview of Clio's interface and architecture (Figure 1).

3.1 Clio Interface

Similar to recent MemDisagg proposals [8, 55], our current implementation adopts a non-transparent interface where applications (running at CNs) allocate and access disaggregated memory via explicit API calls, since doing so gives users opportunities to perform application-specific performance optimizations.[1] Apart from the regular (local) virtual memory address space, each process has a separate ***R**emote virtual memory **A**ddress **S**pace* (*RAS* for short). Each application process has a unique global *PID* across all CNs which is assigned by Clio when the application starts. Overall, programming in RAS is similar to traditional multi-threaded programming except that memory read and write are explicit and that processes running on different CNs can share memory in the same RAS. Figure 2 illustrates the usage of Clio with a simple code example.

[1] CBoard could potentially work with a transparent MemDisagg solution if CNs can directly intercept memory instructions and issue remote requests using application VAs (*e.g.*, with LegoOS's processor architecture).

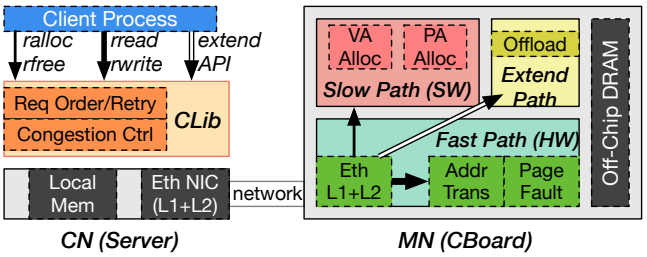

Figure 1: **Clio Architecture.**

```
1  /* Alloc one remote page. Define a remote lock */
2  #define PAGE_SIZE (1<<22)
3  void *remote_addr = ralloc(PAGE_SIZE);
4  ras_lock lock;
5
6  /* Acquire lock to enter critical section.
7     Do two AYSNC writes then poll completion. */
8  void thread1(void *) {
9   rlock(lock);
10  e[0]=rwrite(remote_addr, local_wbuf1,len, ASYNC);
11  e[1]=rwrite(remote_addr+len, local_wbuf2,len, ASYNC);
12  runlock(lock);
13  rpoll(e, 2);
14 }
15
16 /* Synchronously read from remote */
17 void thread2(void *) {
18  rlock(lock);
19  rread(remote_addr, local_rbuf, len, SYNC);
20  runlock(lock);
21 }
```

Figure 2: **Example of Using Clio.**

An application process can perform a set of virtual memory operations in its RAS, including `ralloc`, `rfree`, `rread`, `rwrite`, and a set of atomic and synchronization primitives (*e.g.*, `rlock`, `runlock`, `rfence`). `ralloc` works like `malloc` and returns a VA in RAS, and `rread` and `rwrite` can then be issued to any allocated VAs. As with the traditional virtual memory interface, an RAS has a byte granularity for allocation and access. For `ralloc`, `rfree`, `rread`, and `rwrite`, we offer two versions, synchronous and asynchronous, for users to choose between performance and consistency levels. A synchronous API blocks until the result is ready. An asynchronous API is non-blocking, and the application calls `rpoll` to get its result.

Intra-thread request ordering. Within a thread, synchronous APIs follow strict ordering, and asynchronous APIs follow a release order. Specifically, asynchronous operations may be executed out of order as long as 1) all asynchronous operations before an `rrelease` complete before the `rrelease` returns, and 2) `rrelease` operations are strictly ordered. On top of this release order, we check read/write dependencies (WAR, RAW, WAW) for asynchronous operations targeting the same page. The resulting memory consistency level is the same as architectures like ARMv8 [10]. In addition, we also ensure consistency between metadata and data operations, by ensuring that potentially conflicting operations execute synchronously in the program order. For example, if there is an ongoing `rfree`

request to a VA range, no read or write to that range can start until the `rfree` finishes. Finally, failed or unresponsive requests are transparently retried, and they follow the same ordering guarantees.

Thread synchronization and data coherence. Threads and processes can share data (even when they are not on the same CN). Similar to traditional concurrent programming, Clio threads can use synchronization primitives to build critical sections (*e.g.*, with `rlock`) and other synchronization mechanisms (*e.g.*, flushing all requests with `rfence`). An application can choose to cache data read from `rread` at the CN (*e.g.*, by maintaining `local_rbuf` in the code example). Different processes sharing data in an RAS could have their own cached copies at different CNs. Clio does not make these cached copies coherent automatically and lets applications choose their own coherence mechanisms and policies. We made this deliberate decision because automatic cache coherence on every read/write would incur high performance overhead with commodity Ethernet infrastructure and application semantics could reduce this overhead.

3.2 Clio Architecture

In Clio, CNs are regular servers, each equipped with a regular Ethernet NIC and connected to a ToR switch. MNs are our customized devices directly connected to a ToR switch.

Applications run at CNs on top of our library called *CLib*. CLib handles application requests in the user space. It is in charge of request ordering, request retry, and congestion control. Similar to DPDK [18], CLib bypasses kernel and has zero memory copy capability.

By design,[2] an MN in Clio is a CBoard that consists of an ASIC that runs the hardware logic for all data accesses (we call it the *fast path*), an ARM processor which runs software for handling metadata and control operations (*i.e.*, the *slow path*), and an FPGA that hosts application computation offloading (*i.e.*, the *extend path*). An incoming request arrives at the ASIC and travels through standard Ethernet physical and MAC layers and a Match-and-Action-Table (MAT) that decides which of the three paths the request should go to based on the request type. If the request is a data access (fast path), it stays in the ASIC and goes through a hardware-based virtual memory system that performs three tasks in the same pipeline: address translation, permission checking, and page fault handling (if any). Afterwards, the actual memory access is performed through the memory controller, and the response is formed and sent out through the network stack. Metadata operations such as memory allocation are sent to the slow path. Finally, requests with customized, high-level operations such as pointer chasing and offloaded computation are handled in the FPGA extend path. In the rest of the paper, we focus on the design of the fast and the slow paths and how they interact with each other.

[2] We prototyped CBoard's ASIC part with an FPGA (§5).

4 Clio Design

This section presents the design challenges of building a hardware-based MemDisagg system and our solutions.

4.1 Design Challenges and Principles

Building a hardware-based MemDisagg platform is a previously unexplored area and introduces new challenges mainly because of hardware's restrictions and the unique requirements of MemDisagg.

Challenge 1: The hardware should avoid maintaining or processing complex data structures, because unlike software, hardware has limited resources such as on-chip memory and logic cells. For example, Linux and many other software systems use trees (*e.g.*, the vma tree) for allocation. Maintaining and searching a big tree data structure in hardware, however, would require huge on-chip memory and many logic cells to perform the look up operation (or alternatively use less resources but suffer from performance loss).

Challenge 2: States and buffers that the hardware uses should have bounded sizes, so that they can be statically planned and fit into the on-chip memory. Although swapping between on-chip and off-chip memory is possible, doing so would increase both tail latency and hardware logic complexity. Thus, it is desirable to resort as little as possible to swapping. Achieving the bounded buffer/state goal is even harder when we simultaneously need to meet our scalability goals. Unfortunately, traditional software approaches involve many states and buffers that are large and unscalable. For example, today's reliable network transport layer maintains per-connection sequence numbers and buffer unacknowledged packets for packet ordering and retransmission, and they grow with the number of connections.

Challenge 3: The hardware pipeline should be deterministic and smooth, *i.e.*, it uses a bounded, known number of cycles to process a data unit, and for each cycle, the pipeline can take in one new data unit (from the network). The former would ensure low tail latency, while the latter would guarantee a throughput that could match network line rate. Another subtle benefit of a deterministic pipeline is that we can know the maximum time a data unit stays at MN, which could help bound the size of certain buffers (*e.g.*, §4.5). However, many traditional hardware solutions are not designed to be deterministic and smooth, and we cannot directly adopt their approaches. For example, traditional CPU pipelines could have stalls because of data hazards and have non-deterministic latency to handle memory instructions.

To confront these challenges, we took a clean-slate approach by designing Clio's virtual memory system and network system to follow the following principles.

Principle 1: Eliminate states whenever possible. Not all states in server-based solutions are necessary if we could redesign the hardware. For example, we get rid of RDMA's MR indirection and its metadata altogether by directly map-

ping from application process RAS VAs to PAs (instead of to MRs then to PAs).

Principle 2: Moving non-critical operations and states to software and making the hardware fast path deterministic. If an operation is non-critical and it involves complex processing logic and/or metadata, our idea is to move it to the software slow path running in an ARM processor. For example, VA allocation (`ralloc`) is expected to be a rare operation because applications know the disaggregated nature and would typically have only a few large allocations during the execution. Handling `ralloc`, however, would involve dealing with complex allocation trees. We thus handle `ralloc` and `rfree` in the software slow path. Furthermore, in order to make the fast path performance deterministic, we decouple all slow-path tasks from the performance critical path by *asynchronously* performing them in the background. Note that page fault is a relatively critical operation, as all first accesses to allocated virtual pages will cause a fault, and applications like serverless computing could access large amounts of (new) memory in a short period of time.

Principle 3: Shifting functionalities, states, and buffers to CNs. While hardware resources are scarce at MNs, CNs have sufficient memory and processing power, and it is faster to develop functionalities in CN software. A viable solution is to shift states and functionalities from MNs to CNs. The key question here is how much and what to shift; shifting too much would make Clio similar to PDM and suffer from various performance and security issues of PDM. Our strategy is to shift functionalities to CNs only if doing so 1) could largely reduce hardware resource consumption at MNs, 2) does not slow down common-case foreground data operations, 3) does not sacrifice security guarantees, and 4) adds bounded memory space and CPU cycle overheads to CNs. As a tradeoff, the shift may result in certain uncommon operations (*e.g.*, handling a failed request) being slower.

Principle 4: Making off-chip data structures efficient and scalable. Principles 1 to 3 allow us to reduce MN hardware to only the most essential functionalities and states. For these states, we store them in off-chip memory and cache a fixed amount of them in on-chip memory. Different from most caching solutions, our focus is to make the access to off-chip data structure fast and scalable, *i.e.*, all cache misses have bounded latency regardless of the number of client processes accessing an MN or the size of memory the MN hosts.

Principle 5: Making the hardware fast path smooth by treating each data unit independently at MN. If data units have dependencies (*e.g.*, must be executed in a certain order), then the fast path cannot always execute a data unit when receiving it. To handle one data unit per cycle and reach network line rate, we make each data unit independent by including all the information needed to process a unit in it and by allowing MNs to execute data units in any order that they arrive. To deliver our consistency guarantees, we opt for enforcing request ordering at CNs before sending them out.

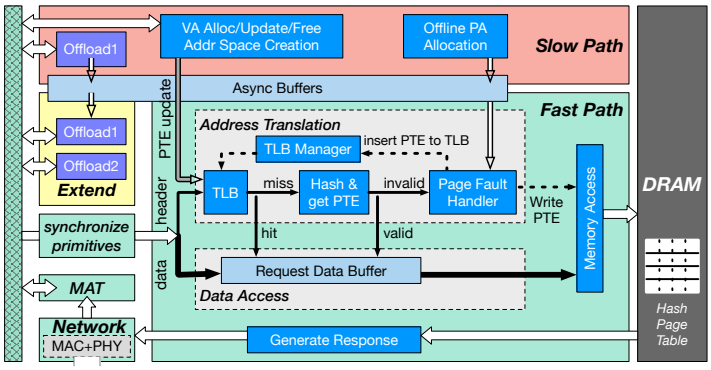

Figure 3: **CBoard Design.** *Green, yellow, and red areas are anticipated to be built with ASIC, FPGA, and low-power cores.*

The rest of this section presents how we follow these principles to design Clio's three main functionalities: memory address translation and protection, page fault handling, and networking. We also briefly discuss our offloading support.

4.2 Scalable, Fast Address Translation

Similar to traditional virtual memory system, we use fix-size pages as address allocation and translation unit, while data is byte-addressable. Despite the similarity in the goal of address translation, the radix-tree-style, per-address space page table design used by all current architectures [60] do not fit MemDisagg for two reasons. First, each request from the network could be from a different client process. If each process has its own page table, MN would need to cache and look up the page table root, causing additional overhead. Second, a multi-level page table design may require multiple DRAM accesses when there is a TLB miss [73]. TLB misses will be much more common in a MemDisagg environment, since with more applications sharing an MN, the total working set size is much bigger than that in a single-server setting, while the TLB size in an MN will be similar or even smaller than a single server's TLB (for cost concerns). To make matters worse, each DRAM access is more costly for systems like RDMA NIC which has to cross the PCIe bus to access the page table in main memory [47, 64].

Flat, single page table design (Principle 4). We propose a new *overflow-free* hash-based page table design that sets the total page table size according to the physical memory size and bounds *address translation to at most one DRAM access*. Specifically, we store *all* page table entries (PTEs) from all processes in a single hash table whose size is proportional to the physical memory size of an MN. The location of this page table is fixed in the off-chip DRAM and is known by the fast path address translation unit, thus avoiding a lookup. As we anticipate applications to allocate big chunks of VAs in their RAS, we use huge pages and support a configurable set of page sizes. With 4 MB page size, the hash table consumes only 0.4% of the physical memory.

The hash value of a VA and its PID are used as the index to determine which hash bucket the corresponding PTE goes to.

Each hash bucket has a fixed number of (K) slots. To access the page table, we always fetch the entire bucket including all K slots using a single DRAM access.

A well-known problem with hash-based page table design is hash collisions that could overflowing a bucket. Existing hash-based page table designs rely on collision chaining [12] or open addressing [73] to handle overflows, both require multiple DRAM accesses or even costly software intervention. In order to bound address translation to at most one DRAM access, we use a novel technique to avoid hash overflows at *VA allocation time*.

VA allocation (Principle 2). The slow path software handles `ralloc` requests and allocate VA. The software allocator maintains a per-process VA allocation tree that records allocated VA ranges and permissions, similar to the Linux vma tree [35]. To allocate size k of VAs, it first finds an available address range of size k in the tree. It then calculates the hash values of the virtual pages in this address range and checks if inserting them to the page table would cause any hash overflow. If so, it does another search for available VAs. These steps repeat until it finds a valid VA range.

Our design trades potential retry overhead at allocation time (at slow path) for better run-time performance and simpler hardware design (at fast path). This overhead is manageable because 1) each retry takes only a few microseconds with our implementation, 2) we employ huge pages, which means fewer pages need to be allocated, 3) we choose a hash function that has very low collision rate [70], and 4) we set the page table to have extra slots ($2\times$ by default). We find no conflicts when memory is below half utilized and has only up to 60 retries when memory is close to full (Figure 13).

TLB. Clio implements a TLB in a fix-sized on-chip memory area and looks it up using content-addressable-memory in the fast path. On a TLB miss, the fast path fetches the PTE from off-chip memory and inserts it to the TLB by replacing an existing TLB entry with the LRU policy. When updating a PTE, the fast path also updates the TLB, in a way that ensures the consistency of inflight operations.

Limitation. A downside of our overflow-free VA allocation design is that it cannot guarantee that a specific VA can be inserted to the page table. This is not a problem for regular VA allocation but could be problematic for allocations that require a fixed VA (*e.g.*, `mmap(MAP_FIXED)`). Currently, Clio finds a new VA range if the user-specified range cannot be inserted to the page table. Applications that must map at fixed VAs (*e.g.*, libraries) will need to use local memory.

4.3 Low-Tail-Latency Page Fault Handling

Page faults are traditionally signaled by the hardware and handled by the OS, and they can happen when a PTE is invalid (VA created, PA not allocated) or when there is a permission violation. While the latter is uncommon, the former happens at every initial access to a VA and could be common (*e.g.*, serverless computing and microservices both frequently start many short running processes, incurring many initial-access page faults). Unfortunately, today's page fault handling mechanism is slow because of the costly interrupt and trap-to-kernel process. For example, a remote page fault via RDMA costs $16.8\,ms$. To avoid page faults, most RDMA-based system pre-allocate big chunks of physical memory and pin them physically. However, doing so results in memory wastes and makes it hard for an MN to pack more applications.

We propose to *handle page faults in hardware and with bounded latency*—a *constant three cycles* to be more specific with our implementation of CBoard. Achieving this performance is not easy. While handling permission-violation faults in hardware is easy (just by sending an error message as the request response), handling initial-access faults in hardware is challenging, as initial accesses require PA allocation, which is a complex operation that involves manipulating complex data structures. Thus, PA allocation should be performed by the slow path (**Challenge 1**). However, if the fast-path page fault handler has to wait for the slow path to generate a PA for each page fault, it will be slow.

To solve this problem, we propose an asynchronous design to shift PA allocation off the performance critical path (**Principle 2**). Specifically, we maintain a set of *free physical page numbers* in an *async buffer*, which the ARM continuously fulfills by finding free physical page addresses and reserving them without actually using the pages. During a page fault, the page fault handler simply fetches a pre-allocated physical page address. Note that even though a single PA allocation operation has a non-trivial delay, the throughput of generating PAs and filling the async buffer is higher than network line rate. Thus, the fast path can always find free PAs in the async buffer in time. After getting a PA from the async buffer and establishing a valid PTE, the page fault handler performs three tasks in parallel: writing the PTE to the off-chip page table, inserting the PTE to the TLB, and continuing the original faulting request. This parallel design hides the performance overhead of the first two tasks, allowing foreground request to proceed immediately.

A recent work [37] also handles page fault in hardware. Its focus is on the complex interaction with kernel and storage devices, and it is a simulation-only work. Clio uses a different design for handling page fault in hardware with the goal of low tail latency, and we built it in real hardware.

Putting virtual memory system together. We illustrate how CBoard's virtual memory system works using a simple example of allocating some memory and writing to it. The first step (`ralloc`) is handled by the slow path, which allocates a VA range by finding an available set of slots in the hash page table. The slow path forwards the new PTEs to the fast path, which inserts them to the page table. At this point, the PTEs are invalid. This VA range is returned to the client. When the client performs the first write, the request goes to the fast path. There will be a TLB miss, followed by a fetch

of the PTE. Since the PTE is invalid, the page fault handler will be triggered, which fetches a free PA from the async buffer and establishes the valid PTE. It will then execute the write, update the page table, and insert the PTE to TLB.

4.4 Stateless and Bufferless MN Network

With huge amounts of research and development efforts, today's data-center network systems are highly optimized in their performance. Our goal of Clio's network system is unique and fit MemDisagg's requirements—minimizing the network stack's hardware resource consumption at MNs and achieving great scalability, while maintaining similar performance as today's fast network. Traditional software-based reliable transports like Linux TCP incurs high performance overhead. Today's hardware-based reliable transports like RDMA are fast, but they require a fair amount of (on-chip) memory to maintain states, *e.g.*, per-connection sequence numbers, congestion states [9], or bitmaps [42, 45], not meeting our low-cost goal.

Our insight is that different from general-purpose network communication where each endpoint can be both the sender (requester) and the receiver (responder) that exchange general-purpose messages, MNs only respond to requests sent by CNs, and these requests are all memory-related operations that have their specific properties. Based this insight, we design a new network system with two main ideas. Our first idea is to maintain transport logic, states, and buffers only at CNs, essentially making MNs "transportless" and thus *stateless* and *bufferless* (**Principle 3**)[3]. Our second idea is to relax the reliability of the transport and instead enforce ordering and loss recovery at the memory request level, so that MNs' hardware pipeline can process data units as soon as they arrive (**Principle 5**).

With these ideas, we implemented a transport in CLib at CNs. CLib bypasses the kernel to directly issue raw Ethernet requests to a regular Ethernet NIC. The MN includes only standard Ethernet physical and link layers and a slim ack-generation layer (§5). We now describe our detail design.

Removing connections with request-response semantics. Connections (*i.e.*, QPs) are a major scalability issue with RDMA. Similar to recent works [46, 59], we make our network system connection-less with memory request-response pairs. Applications running at CNs directly initiate Clio APIs to an MN without any connections. CLib uses the response of each Clio request as the ACK and matches it to the request using a request ID.

Lifting reliability to the memory request level. Instead of triggering a retransmission protocol for every lost/corrupted packet at the transport layer, CLib retries the entire memory request if any packet is lost or corrupted (either in the sending or the receiving direction). On the receiving path, MN's network stack only checks packet's integrity. If a packet is corrupted, the MN immediately sends a NACK to the sender CN. CLib retries a memory request if one of three situations happen: a NACK is received, the response from MN is corrupted, or no response is received within a dynamic retransmission timeout (RTO) period. The RTO is computed using the moving average of prior end-to-end RTTs. In addition to lifting retransmission from transport to the request level, we also lift ordering to the memory request level and allow out-of-order packet delivery (see details in §4.5).

CN-managed congestion and incast control. Our goal of controlling congestion in the network and handling incast that can happen both at a CN and an MN is to *eliminate* states at MN. To this end, we build the entire congestion and incast control at CN in the CLib. To control congestion, CLib uses a simple delay-based, reactive policy that uses end-to-end RTT delay as the congestion signal, similar to recent sender-managed, delay-based mechanisms [36, 44, 59]. Each CN maintains one congestion window, *cwnd*, per MN that controls the maximum number of outstanding requests that can be made to the MN from this CN. We adjust *cwnd* based on measured delay using a standard Additive Increase Multiplicative Decrease (AIMD) manner.

To handle incast to a CN, we exploit the fact that the CN knows the sizes of expected responses for the requests that it sends out and that responses are the major incoming traffic to it. Each CLib maintains one incast window, *iwnd*, and sends a request only when both *cwnd* and *iwnd* have room.

Handling incast to an MN at CNs is more challenging, as we cannot throttle incoming traffic at the MN side or would otherwise maintain states at MNs. To handle MN incast at CNs, we draw inspiration from Swift [36] by allowing *cwnd* to fall below one packet when long delay is observed at a CN. For example, a *cwnd* of 0.1 means that the CN can only send a packet within 10 RTTs.

4.5 Request Ordering and Data Consistency

Allowing intra-request packet re-ordering. Enforcing packet ordering above the link layer normally requires maintaining states (*e.g.*, packet sequence ID) at both the sender and the receiver. To avoid maintaining such states at MNs, our approach is to deal with packet reordering only at CNs in CLib (**Principle 3**). Specifically, CLib splits a request that is bigger than MTU into several link-layer packets and attaches a Clio header to each packet, which includes sender-receiver addresses, a request ID, and request type. This enables the MN to treat each packet independently (**Principle 5**). It executes packets as soon as they arrive, even if they are not in the sending order. This out-of-order data placement semantic is in line with RDMA specification [45]. Note that only write requests will be bigger than MTU, and the order of data writing within a write request does not affect correctness as long as proper inter-request ordering is followed.

[3] 1RMA [59], a recent server-based remote memory system, onloads most of its retransmission and congestion logic from the NIC to the host CPU. As a result, 1RMA's NIC is simple. However, 1RMA relies on a companion host to function, violating the "server-less" goal of MNs.

When a CN receives multiple link-layer packets belonging to the same request response, CLib reassembles them before delivering to the application.

Enforcing intra-thread inter-request ordering at CN based on memory semantics. As explained in §3.1, Clio supports both synchronous and asynchronous APIs. Since only one synchronous request can be outstanding in a thread, there cannot be any inter-request reordering problem. On the other hand, asynchronous requests require proper ordering since there can be multiple outstanding asynchronous requests and we need to enforce proper dependency (WAW, RAW, and WAR) and flush requests before `rrelease`. We enforce these ordering requirements at CNs in CLib instead of at MNs (**Principle 3**) for two reasons. First, enforcing ordering at MNs requires more on-chip memory and complex logic in hardware. Second, even if we enforce ordering at MNs, network reordering would still break end-to-end ordering guarantees, and we relax network ordering guarantees to minimize the hardware resource consumption at MNs (§4.4).

Specifically, CLib keeps track of all inflight requests and matches every new request's virtual page number (VPN) to the inflight ones'. If a WAR, RAW, or WAW dependency is detected, CLib blocks the new request until the conflicting request finishes. When CLib sees a `rrelease` operation, it waits until all inflight requests return or time out. We currently track dependencies at the page granularity mainly to reduce tracking complexity and metadata overhead. The down side is that false dependencies could happen (*e.g.*, two accesses to the same page but different addresses). False dependencies could be reduced by dynamically adapting the tracking granularity if application access patterns are tracked—we leave this improvement for future work.

Inter-thread/process consistency. Multi-threaded or multi-process concurrent programming on Clio could use the synchronization primitives Clio provides to ensure data consistency (§3.1). We implemented all synchronization primitives like `rlock` and `rfence` at MN, because they need to work across threads and processes that possibly reside on different CNs. Before a request enters either the fast or the slow paths, MN checks if it is a synchronization primitive. For primitives like `rlock` that internally is implemented using atomic operations like `TAS`, MN blocks future atomic operations until the current one completes. For `rfence`, MN blocks all future requests until all inflight ones complete. Inflight synchronization primitives are one of the only two cases where MN needs to maintain states. As these operations are infrequent and each operation executes in bounded time, the hardware resources for maintaining these states are minimal and bounded.

Idempotence. In Clio, most types of requests like `rread` and `rwrite` are idempotent and can be retried multiple times with the same result. Clio also includes a few types of non-idempotent requests such as atomic increment. To ensure that retrying non-idempotent requests will not generate wrong results, we maintain a small ring buffer at MN to record the request IDs and results of K recently executed non-idempotent requests. If MN receives a request with the same ID in the buffer, it will not execute it and directly send the cached result as the response. An MN will not run any new non-idempotent requests when the ring buffer is full. To properly maintain the ring buffer, CNs send an ACK back to MN after receiving the response from it. This ACK will free a slot in the ring buffer. As non-idempotent requests are rare, MN only needs to maintain a small K-sized buffer, and this is the one of the only two cases where MN maintains states.

4.6 Extension and Offloading Support

To avoid network round trips when working with complex data structures and/or performing data-intensive operations, we extend the core MN to support high-level APIs and application computation offloading in the extend path, which includes an FPGA chip and the ARM processor. We only have space to give a high-level overview of the extend path, leaving details to a follow-on paper. Users can write and deploy application offloads both in FPGA and in software. An offload can either be the handler of a high-level API (*e.g.*, pointer chasing) or an entire function (*e.g.*, data filtering). To ease the development of offloads, Clio offers the same virtual memory interface as the one to applications running at CNs. Each offload has its own address space, and it could also share data with processes running at CNs. Developing offloads is thus closer to traditional multi-threaded programming (in terms of memory accesses).

5 CBoard Prototyping

We prototyped CBoard with a low-cost Xilinx MPSoC board [72] and build the hardware fast path (which is anticipated to be built in ASIC) with FPGA. This board consists of a small FPGA with 504K logic cells (LUTs) and 4.75 MB FPGA memory (BRAM), a quad-core ARM Cortex-A53 processor, two 10 Gbps SFP+ ports connected to the FPGA, and 2 GB of off-chip on-board memory. This board has several differences from our anticipated real CBoard: its network port bandwidth and on-board memory size are both much lower than our projection, and like all FPGA prototypes, its clock frequency is much lower than real ASIC. Unfortunately, no board on the market offers the combination of small FPGA/ARM (required for low cost) with large memory and high-speed network ports.

Nonetheless, certain features of this board are likely to exist in a real CBoard, and these features guide our implementation. Its ARM processor and the FPGA connect through an interconnect that has high bandwidth (90 GB/s) but high delay (40 μs). Although better interconnects could be built, crossing ARM and FPGA would inevitably incur non-trivial performance overhead. With this board, both the ARM and the FPGA can access an on-board DRAM through a DDR interface, but the ARM's access is much slower because it has

to first physically cross the FPGA then to the DRAM. A better design would connect the ARM directly to the DRAM, but it will still be slower for the ARM to access on-board DRAM than its local on-chip memory.

Currently, the FPGA prototype of Clio (excluding computation offloads and third-party IPs[4]) includes a total of 5.6K SLOC written in SpinalHDL [61], 2K in C HLS, and 17K in C for host and ARM. All Clio's FPGA modules run at 250 MHz clock frequency and 512-bit data width. They all achieve an *Initiation Interval* (*II*) of one (II is the number of clock cycles between the start time of consecutive loop iterations, and it decides the maximum achievable throughput). Achieving an II of one is not easy and requires careful pipeline design in all the modules.

Below, we pick some techniques used in our prototyping implementation that will still be applicable in a real CBoard.

To mitigate the problem of slow accesses to on-board DRAM from ARM, we maintain shadow copies of metadata at ARM's local DRAM. For example, we store a *shadow* version of the page table in ARM's local memory, and keep it in sync with the real page table in the on-board DRAM. We employ an efficient polling mechanism for ARM/FPGA communication. We dedicate one ARM core to busy poll an RX ring buffer between ARM and FPGA, where the FPGA posts tasks for ARM. This polling thread hands over tasks to other worker threads for task handling and post responses to a TX ring buffer.

CBoard's network stack builds on top of standard, vendor-supplied Ethernet physical and link layer IPs, with just an additional thin checksum-verify and ack-generation layer on top. This layer uses much less resources compared to a normal RDMA-like stack (§7.3). We use lossless Ethernet with Priority Flow Control (PFC) for less packet loss and retransmission. Since PFC has issues like head-of-line blocking [23, 38, 45, 74], we rely on our congestion and incast control to avoid triggering PFC as much as possible.

To assist Clio users in building their applications, we implemented a simple software simulator of CBoard which works with CLib for developers to test their code without the need to run an actual CBoard.

6 Building Applications on Clio

We built three applications on top of Clio, one that uses the basic Clio APIs, one that uses a high-level, extended API, and one that offloads computation to MNs.

Image compression. We build a simple image compression/decompression utility that runs purely at CN. Each client of the utility (*e.g.*, a Facebook user) has its own collection of photos, stored in two arrays at MN, one for compressed and one for original. Because clients' photos need to be protected from each other, we use one process per client to run the utility. The utility simply reads a photo from MN using `rread`, compresses/decompresses it, and writes it back to the other array using `rwrite`. We implemented this utility with 1K C code in 3 developer days.

Radix tree. To demonstrate how to build a data structure on Clio using Clio's extended API, we built a radix tree with linked lists and pointers. Data-structure-level systems like AIFM [55] could follow this example to make simple changes to run on Clio. At MN, we built an extended pointer-chasing API in hardware to perform a value comparison at each chased node and returns either when there is a match or the next pointer becomes null. Searching the radix tree mainly involves going through layers of nodes and performing this modified pointer chasing API. We implemented the radix tree with 300 C code and 150 SpinalHDL code at MN in less than one developer day.

Key-value store. We built *Clio-KV*, a key-value store that supports concurrent create/update/read/delete key-value entries with atomic write and read committed consistency guarantees. Clio-KV runs at MN as computation offloads. Users can access it through a key-value interface from CNs. The Clio-KV module has its own address space. It stores key-value pair data and a chained hash table for metadata in this virtual memory address space and accesses them with Clio virtual memory APIs. We implemented Clio-KV with 772 SpinalHDL code in 6 developer days.

7 Evaluation

Our evaluation reveals the scalability, throughput, median and tail latency, energy and resource consumption of Clio. We compare Clio's end-to-end performance with industry-grade NICs (ASIC) and well-tuned RDMA-based software systems. All Clio's results are FPGA-based, which would be improved with ASIC implementation. Nonetheless, Clio significantly outperforms RDMA on scalability and tail latency, while being similar on other measurements.

Environment. We evaluated Clio on our local cluster of four CNs and one Xilinx ZCU106 board, all connected to a Nvidia 40 Gbps VPI switch. Each CN is a Dell PowerEdge R740 server equipped with a Xeon Gold 5128 CPU and a 40 Gbps Nvidia Connect-X3 NIC, with two of them also have a Nvidia BlueField SmartNIC [43]. We also include results from CloudLab [14] with the Nvidia Connect-X5 NIC.

7.1 Basic Microbenchmark Performance

Scalability. We first compare the scalability of Clio and RDMA. Figure 4 measures the latency of Clio and RDMA as the number of client processes increases. For RDMA, each process uses its own QP. Since Clio is connectionless, it scales perfectly with the number of processes. RDMA scales poorly with its QP, and the problem persist with newer generations of RNIC, which is also confirmed by others [50, 64].

Figure 5 evaluates the scalability with respect to PTEs and memory regions. For the memory region test, we register

[4]We use several third-party IPs: 1) vendor-supplied interconnect and DDR IPs, 2) an open-source MAC and PHY network stack [22].

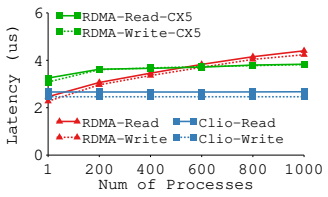

Figure 4: **Process (Connection) Scalability.**

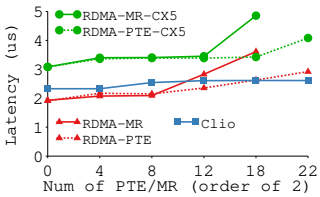

Figure 5: **PTE and MR Scalability.** *RDMA fails beyond 2^{18} MRs.*

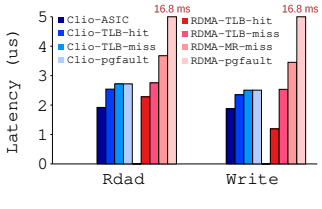

Figure 6: **Comparison of TLB Miss and page fault.** *Clio-ASIC are projected values of TLB hit.*

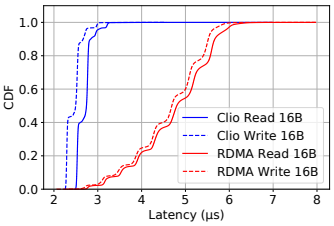

Figure 7: **Latency CDF.**

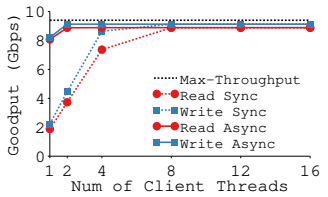

Figure 8: **End-to-End Goodput.** *1 KB requests.*

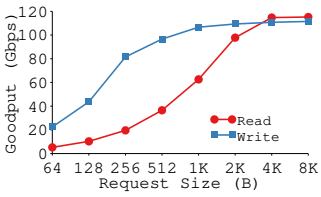

Figure 9: **On-board Goodput.** *FPGA test module generates requests at maximum speed.*

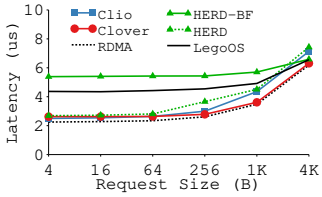

Figure 10: **Read Latency.** *HERD-BF: HERD running on BlueField SmartNIC.*

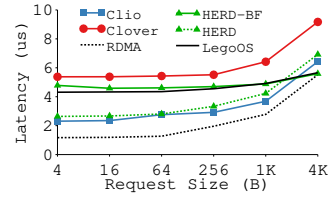

Figure 11: **Write Latency.** *Clover requires at least 2 RTTs for write.*

multiple MRs using the same physical memory for RDMA. For Clio (which gets rid of the MR concept), we use multiple processes to share the same memory, resulting in one PTE per process. RDMA's performance starts to degrade when there are more than 2^8 (local cluster) or 2^{12} (CloudLab), and the scalability wrt MR is worse than wrt PTE. In fact, RDMA fails to run beyond 2^{18} MRs. In contrast, Clio scales well and never fails (at least up to 4 TB memory[5]). It has two levels of latency: lower latency below 2^4 for TLB hit and higher above 2^4 for TLB miss (which results in a constant of one DRAM access). A real CBoard could use a larger TLB if optimal performance is desired.

These experiments confirm that **Clio can handle thousands of concurrent clients and TBs of memory**.

Latency variation. Figure 6 plots the latency of reading/writing 16 B data when the operation results in a TLB hit, a TLB miss, a first-access page fault, and MR miss (for RDMA only). RDMA's performance degrades significantly with misses. Its page fault handling cost is startling high — 16.8 ms. We confirm the same effect on CloudLab. Clio only incurs a small TLB miss cost and **no additional cost of page fault handling**.

We also include a projection of Clio's latency if it was to be implemented using a real ASIC-based CBoard. Specifically, we collect the latency breakdown of time spent on the network wire and at CN, number of cycles on FPGA, and time on accessing on-board DRAM. We maintain the first part, scale the FPGA part to ASIC's frequency (2 GHz), use DDR access time collected on our server to replace the access time to on-board DRAM (which goes through a slow board memory controller). Our projected read latency is better than RDMA, while write is worse. We suspect that it is due to Nvidia RNIC's optimization of replying to a write before it is written to DRAM, which Clio could also adopt.

Figure 7 plots the request latency CDF of continuously running read/write 16 B data. Clearly, Clio has much less latency variation and a much shorter tail than RDMA.

Read/write throughput. We measure Clio's throughput by varying the number of concurrent client threads (Figure 8). Clio's default asynchronous APIs quickly reach the line rate of our testbed (9.4 Gbps maximum throughput). Its synchronous APIs could also reach line rate fairly quickly.

Figure 9 measures the maximum throughput Clio's FPGA implementation could reach without the bottleneck of the board's 10 Gbps port. Both read and write can reach more than 110 Gbps when request size is large. Read throughput is lower than write when request size is smaller. We discovered that the throughput bottleneck is at a third-party non-pipelined DMA IP (which could potentially be improved).

Comparison with other systems. We compare Clio with native one-sided RDMA, Clover [65], HERD [34], and LegoOS [56]. We ran HERD on both CPU and BlueField (HERD-BF). Clover is a passive disaggregated persistent memory system that we adapted as a passive disaggregated memory (PDM) system. HERD is an RDMA-based system that supports a key-value interface with an RPC-like architecture. LegoOS emulates MNs using regular servers and builds its virtual memory system in software.

Clio's performance is similar to HERD and close to native RDMA. Clover's write is the worst because of it uses at least 2 RTTs for writes to deliver its consistency guarantees without any processing power at MNs. HERD-BF's latency is much higher than when HERD runs on CPU due to the slow communication between BlueField's Connect-X5 chip and ARM processor chip. LegoOS's latency is almost two

[5]This calculation is based on the number of PTEs, while our experiments map all of them to a small range of memory as our testbed only has 2 GB physical memory.

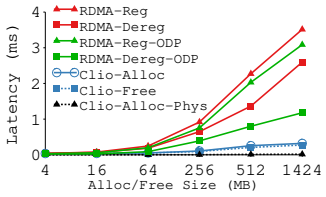

Figure 12: **Alloc/Free Latency.**

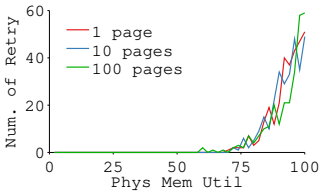

Figure 13: **Alloc Retry Rate.**

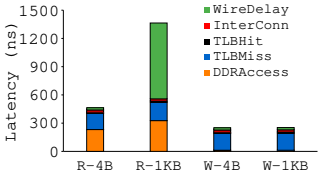

Figure 14: **Latency Breakdown.** *Breakdown of time spent at CBoard.*

Figure 15: **Image Compression.**

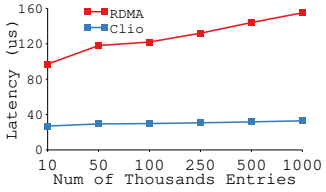

Figure 16: **Radix Tree Search Latency.**

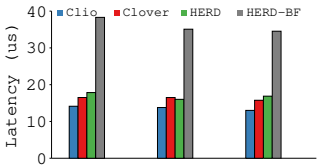

Figure 17: **Key-Value Store YCSB Latency.**

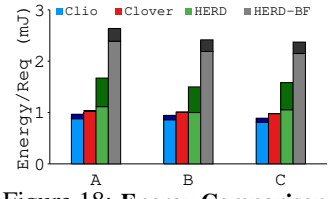

Figure 18: **Energy Comparison.** *Darker/lighter shades represent energy spent at MNs and CNs.*

System/Module	Logic (LUT)	Memory (BRAM)
StRoM-RoCEv2-10G	39%	76%
Tonic-SACK-100G	48%	40%
Clio (Total)	30%	31%
VirtMem	4.8%	3%
NetStack	2.3%	1.7%
Go-Back-N	5.8%	2.6%

Figure 19: **FPGA Utilization.**

times higher than Clio's when request size is small. In addition, from our experiment, LegoOS can only reach a peak throughput of 77 Gbps, while Clio can reach 110 Gbps. LegoOS' performance overhead comes from LegoOS' software approach, demonstrating the necessity of a hardware-based solution like Clio.

Allocation performance. Figure 12 shows Clio's VA and PA allocation comparing to RDMA's MR registration performance (ODP means RDMA is in On-Demand-Paging mode). Clio's PA allocation takes less than 20 μs, and the VA allocation is much faster than RDMA MR registration, although both get slower with larger allocation/registration size. Figure 13 shows the number of retries at allocation time with three alloc sizes (1, 10, and 100 pages) as the physical memory gets filled up. There is no retry when memory is below half utilized. Even when memory is close to full, there are at most 60 retries per allocation request, with roughly 0.5 ms per retry. This confirms that our design of avoiding hash overflows at allocation time is practical.

Close look at CBoard components. To further understand Clio's performance, we profile different parts of Clio's processing for read and write of 4 B to 1 KB. CLib adds a very small overhead (250 ns in total), thanks to our efficient threading model and network stack implementation. Figure 14 shows the latency breakdown at CBoard. Time to fetch data from DRAM (DDRAccess) and to transfer it over the wire (WireDelay) are the main contributor to read latency, especially with large read size. Both could be largely improved in a real CBoard with better memory controller and higher frequency. TLB miss (which takes one DRAM read) is the other main part of all the latencies.

7.2 Application Performance

Image Compression. We run a workload where each client compresses and decompresses 1000 256*256-pixel images with increasing number of concurrently running clients. Figure 15 shows the total runtime per client. Clio's performance stays the same as the number of clients increase. RDMA's performance does not scale because it requires each client to register a different MR.

Radix Tree. Figure 16 shows the latency of searching a key in pre-populated radix trees when varying the tree size. RDMA's performance is worse than Clio, because it requires multiple RTTs to traverse the tree, while Clio only needs one RTT for each pointer chasing. Unlike Clio, RDMA's performance also scales poorly.

Key-value store. In Figure 17, we evaluate Clio-KV using the YCSB benchmark [1] and compare it to Clover, HERD, and HERD-BF. We run two CNs and 8 threads per CN. We use 100K key-value entries and run 100K operations per test, with YCSB's default key-value size of 1 KB. The accesses to keys follow the Zipf distribution ($\theta = 0.99$). We use three YCSB workloads with different *get-set* ratios: 100% *get* (workload C), 5% *set* (B), and 50% *set* (A). Clio-KV outperforms all the other systems.

7.3 Energy Cost and FPGA Utilization

We measure the total energy used for running YCSB workloads by collecting the total CPU (or FPGA) cycles and the Watt of a CPU core [2], ARM processor [53], and FPGA (measured). We omit the energy used by DRAM and NICs in all the calculation. Clover, a system that centers its design around low cost, has slightly higher energy than Clio. Even though there is no processing at MNs for Clover, its CNs use more cycles to process and manage memory. HERD consumes 1.6× to 3× more energy than Clio, mainly because its CPU overhead at MNs. Surprisingly, HERD-BF consumes the most energy, even though it is a low-power ARM-based SmartNIC. This is because of its worse performance and longer total runtime.

Figure 19 compares the FPGA utilization among Clio, StRoM's RoCEv2 [58], and Tonic's selective ack stack [9]. With our design that is tailored to save resources, Clio consumes roughly one third of the total resources. Both StRoM and Tonic include just a network stack yet they consume more resources than Clio. Within Clio, the virtual memory (VirtMem) and the network stack (NetStack) consume a small fraction of the total resources, with the rest being vendor IPs (PHY, MAC, DDR4, and interconnect). To put things in perspective, we implement a RDMA-like Go-back-N network stack which supports 1K connections. It uses 2.5× more logic than what our current network stack consumes. In all, our efficient hardware implementation leaves most FPGA resources available for application offloads.

8 Conclusion

We presented Clio, a new hardware-based disaggregated memory system. Our FPGA prototype demonstrates that Clio achieves great performance, scalability, and cost saving. This work not only helps the future development of MemDisagg solutions but also demonstrate how to implement a core OS subsystem in the hardware.

References

[1] YCSB Github Repository. https://github.com/brianfrankcooper/YCSB.

[2] Intel Xeon Gold 5128. https://ark.intel.com/content/www/us/en/ark/products/192444/intel-xeon-gold-5218-processor-22m-cache-2-30-ghz.html.

[3] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*, Bertinoro, Italy, May 2019.

[4] Alibaba. "pangu – the high performance distributed file system by alibaba cloud". https://www.alibabacloud.com/blog/pangu-the-high-performance-distributed-file-system-by-alibaba-cloud_594059.

[5] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[6] Amazon. Amazon elastic block store. https://aws.amazon.com/ebs/?nc1=h_ls, 2019.

[7] Amazon. Amazon s3. https://aws.amazon.com/s3/, 2019.

[8] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the Application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '20)*.

[9] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*.

[10] ARMv8. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-a-architecture-2016-additions.

[11] Krste Asanović. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers, February 2014. Keynote talk at the 12th USENIX Conference on File and Storage Technologies (FAST '14).

[12] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[13] Brian Cho and Ergin Seyfe. Taking advantage of a disaggregated storage and compute architecture. In *Spark+AI Summit 2019 (SAIS '19)*, San Francisco, CA, USA, April 2019.

[14] CloudLab. https://www.cloudlab.us/.

[15] CXL Consortium. https://www.computeexpresslink.org/.

[16] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of µs-scale rpcs. ASPLOS '19, 2019.

[17] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[18] DPDK. https://www.dpdk.org/.

[19] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, USA, April 2014.

[20] Facebook. Introducing bryce canyon: Our next-generation storage platform. https://code.fb.com/data-center-engineering/introducing-bryce-canyon-our-next-generation-storage-platform/, 2017.

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

[22] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: An Open-Source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '20)*, Fayetteville,AK, May 2020.

[23] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong

Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.

[24] Gen-Z Consortium. https://genzconsortium.org.

[25] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang Shin. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, Boston, MA, USA, April 2017.

[26] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*.

[27] Hewlett Packard. The Machine: A New Kind of Computer. http://www.hpl.hp.com/research/systems-research/themachine/, 2005.

[28] Hewlett-Packard. Memory Technology Evolution: An Overview of System Memory Technologies the 9th edition, 2010. https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-c00256987.

[29] Hewlett Packard Labs. Memory-Driven Computing. https://www.hpe.com/us/en/newsroom/blog-post/2017/05/memory-driven-computing-explained.html, 2017.

[30] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Nick McKeown, and Changhoon Kim. The nanopu: Redesigning the cpu-network interface to minimize rpc tail latency. *arXiv preprint arXiv:2010.12114*, 2020.

[31] Intel Corporation. Intel Rack Scale Architecture: Faster Service Delivery and Lower TCO. http://www.intel.com/content/www/us/en/architecture-and-technology/intel-rack-scale-architecture.html.

[32] ITRS. International Technology Roadmap for Semiconductors (SIA) 2014 Edition.

[33] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for second-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[34] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '14)*, Chicago, IL, USA, August 2014.

[35] Linux Kernel. Red-black trees (rbtree) in linux. https://www.kernel.org/doc/Documentation/rbtree.txt.

[36] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, 2020.

[37] Gyusun Lee, Wenjing Jin, Wonsuk Song, Jeonghun Gong, Jonghyun Bae, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. A case for hardware-based demand paging. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, 2020.

[38] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*.

[39] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, Austin, Texas, 2009.

[40] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*, New Orleans, LA, USA, February 2012.

[41] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs Using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*, Beijing, China, August 2019.

[42] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Memory efficient loss recovery for hardware-based transport in datacenter. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet'17, 2017.

[43] Mellanox. Bluefield smartnic. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf, 2018.

[44] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. *ACM SIGCOMM Computer Communica-*

tion Review (SIGCOMM '15).

[45] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, 2018.

[46] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*.

[47] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, 2018.

[48] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[49] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*, Porto, Portugal, April 2018.

[50] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. Storm: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR '19)*.

[51] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[52] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Transactions Computer System*, 33(3):7:1–7:55, August 2015.

[53] P. Peng, Y. Mingyu, and X. Weisheng. Running 8-bit dynamic fixed-point convolutional neural network on low-cost arm platforms. In *2017 Chinese Automation Congress (CAC)*, 2017.

[54] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[55] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.

[56] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, October 2018.

[57] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 8th Annual Symposium on Cloud Computing (SOCC '17)*, Santa Clara, CA, USA, September 2017.

[58] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, Heraklion, Greece, April 2020.

[59] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*.

[60] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[61] SpinalHDL. SpinalHDL. https://github.com/SpinalHDL/SpinalHDL.

[62] Jon Tate, Pall Beck, Hector Hugo Ibarra, Shanmuganathan Kumaravel, Libor Miklas, et al. *Introduction to storage area networks*. IBM Redbooks, 2018.

[63] TECHPP. Alibaba singles' day 2019 had a record peak order rate of 544,000 per second. https://techpp.com/2019/11/19/alibaba-singles-day-2019-record/, 2019.

[64] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. Pythia: Remote oracles for the masses. In *28th USENIX Security Symposium (USENIX Security 19)*.

[65] Shin-Yeh Tsai, Yizhou Shan, , and Yiying Zhang. Disaggregating Persistent Memory and Controlling Them from Remote: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of the*

2020 USENIX Annual Technical Conference (ATC '20), Boston, MA, USA, July 2020.

[66] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[67] Haris Volos, Kimberly Keeton, Yupu Zhang, Milind Chabbi, Se Kwon Lee, Mark Lillibridge, Yuvraj Patel, and Wei Zhang. Memory-Oriented Distributed Computing at Rack Scale. In *Proceedings of the ACM Symposium on Cloud Computing, (SoCC '18)*, Carlsbad, CA, USA, October 2018.

[68] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, Santa Clara, CA, February 2020.

[69] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.

[70] Wikipedia. "jenkins hash function". https://en.wikipedia.org/wiki/Jenkins_hash_function.

[71] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1), March 1995.

[72] Xilinx. Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/zcu106.html. Accessed May 2020.

[73] Idan Yaniv and Dan Tsafrir. Hash, don't cache (the page table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, 2016.

[74] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*.