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

Zhiyuan Guo*
University of California, San Diego
San Diego, California, USA
z9guo@ucsd.edu

Yizhou Shan*
University of California, San Diego
San Diego, California, USA
ys@ucsd.edu

Xuhao Luo
University of California, San Diego
San Diego, California, USA
x3luo@ucsd.edu

Yutong Huang
University of California, San Diego
San Diego, California, USA
yutonghuang@ucsd.edu

Yiying Zhang
University of California, San Diego
San Diego, California, USA
yiying@ucsd.edu

ABSTRACT

Memory disaggregation has attracted great attention recently because of its benefits in efficient memory utilization and ease of management. So far, memory disaggregation research has all taken one of two approaches: building/emulating memory nodes using regular servers or building them using raw memory devices with no processing power. The former incurs higher monetary cost and faces tail latency and scalability limitations, while the latter introduces performance, security, and management problems.

Server-based memory nodes and memory nodes with no processing power are two extreme approaches. We seek a sweet spot in the middle by proposing a hardware-based memory disaggregation solution that has the right amount of processing power at memory nodes. Furthermore, we take a clean-slate approach by starting from the requirements of memory disaggregation and designing a *memory-disaggregation-native* system.

We built *Clio*, a disaggregated memory system that virtualizes, protects, and manages disaggregated memory at hardware-based memory nodes. The Clio hardware includes a new virtual memory system, a customized network system, and a framework for computation offloading. In building Clio, we not only co-design OS functionalities, hardware architecture, and the network system, but also co-design compute nodes and memory nodes. Our FPGA prototype of Clio demonstrates that each memory node can achieve 100 Gbps throughput and an end-to-end latency of 2.5 μs at median and 3.2 μs at the 99th percentile. Clio also scales much better and has orders of magnitude lower tail latency than RDMA. It has 1.1× to 3.4× energy saving compared to CPU-based and SmartNIC-based disaggregated memory systems and is 2.7× faster than software-based SmartNIC solutions.

*Both authors contributed equally to the paper

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507762

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Hardware** → *Communication hardware, interfaces and storage*; • **Software and its engineering** → *Virtual memory*.

KEYWORDS

Resource Disaggregation, FPGA, Virtual Memory, Hardware-Software Co-design

ACM Reference Format:
Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3503222.3507762

1 INTRODUCTION

Modern datacenter applications like graph computing, data analytics, and deep learning have an increasing demand for access to large amounts of memory [5]. Unfortunately, servers are facing *memory capacity walls* because of pin, space, and power limitations [30, 33, 81]. 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, datacenters are seeing the needs from management and resource utilization perspectives to *disaggregate* resources [13, 73, 78]—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 researches have all taken one of two approaches: building/emulating MNs using regular servers [5, 26, 53, 63, 64] or using raw memory devices with no processing power [29, 45, 46, 75]. The fundamental issues of server-based approaches such as RDMA-based systems are the monetary and energy 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 [75].

Server-based MNs and MNs with no processing power are two extreme approaches of building MNs. We seek 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 *Clio*[1], a hardware-based disaggregated memory system. 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 can allocate memory from the same CBoard, with each process having its own *remote virtual memory address space*. Furthermore, one remote virtual memory address space can span multiple CBoards. Applications can perform byte-granularity remote memory read/write and use Clio's synchronization primitives for synchronizing concurrent accesses to shared remote memory .

A key research question in designing Clio is ***how to use limited hardware resources to achieve 100 Gbps, microsecond-level average and tail latency for TBs of memory and thousands of concurrent clients?*** These goals are important and unique for MemDisagg. A good MemDisagg solution should reduce the total CapEx and OpEx costs compared to traditional non-disaggregated systems and thus cannot afford to use large amounts of hardware resources at MNs. Meanwhile, remote memory accesses should have high throughput and low average and tail latency, because even after caching data at CN-local memory, there can still be fairly frequent accesses to MNs and the overall application performance can be impacted if they are slow [22]. Finally, unlike traditional single-server memory, a disaggregated MN should allow many CNs to store large amounts of data so that we only need a few of them to reduce costs and connection points in a cluster. How to achieve each of the above cost, performance, and scalability goals *individually* is relatively well understood. However, achieving all these seemingly conflicting goals *simultaneously* is hard and previously unexplored.

Our main idea is to ***eliminate state from the MN hardware***. Here, we overload the term "state elimination" with two meanings: 1) the MN can treat each of its incoming requests in isolation even if requests that the client issues can sometimes be inter-dependent, and 2) the MN hardware does not store metadata or deals with it. Without remembering previous requests or storing metadata, an MN would only need a tiny amount of on-chip memory that does not grow with more clients, thereby *saving monetary and energy cost* and achieving *great scalability*. Moreover, without state, the hardware pipeline can be made *smooth* and *performance deterministic*. A smooth pipeline means that the pipeline does not stall, which is only possible if requests do not need to wait for each other. It can then take one incoming data unit from the network every fixed number of cycles (1 cycle in our implementation), achieving constantly *high throughput*. A performance-deterministic pipeline means that the hardware processing does not need to wait for any slower metadata operations and thus has *bounded tail latency*.

[1]Clio is the daughter of Mnemosyne, the Greek goddess of memory.

Effective as it is, can we really eliminate state from MN hardware? First, as with any memory systems, users of a disaggregate memory system expect it to deliver certain reliability and consistency guarantees (*e.g.*, a successful write should have all its data written to remote memory, a read should not see the intermediate state of a write, etc.). Implementing these guarantees requires proper ordering among requests and involves state even on a single server. The network separation of disaggregated memory would only make matters more complicated. Second, quite a few memory operations involve metadata, and they too need to be supported by disaggregated memory. Finally, many memory and network functionalities are traditionally associated with a client process and involve per-process/client metadata (*e.g.*, one page table per process, one connection per client, etc.). Overcoming these challenges require the re-design of traditional memory and network systems.

Our first approach is to separate the metadata/control plane and the data plane, with the former running as software on a low-power ARM-based SoC at MN and the latter in hardware at MN. Metadata operations like memory allocation usually need more memory but are rarer (thus not as performance critical) compared to data operations. A low-power SoC's computation speed and its local DRAM are sufficient for metadata operations. On the other hand, data operations (*i.e.*, all memory accesses) should be fast and are best handled purely in hardware. Even though the separation of data and control plane is a common technique that has been applied in many areas [25, 39, 61], a separation of memory system control and data planes has not been explored before and is not easy, as we will show in this paper.

Our second approach is to re-design the memory and networking data plane so that most state can be managed only at the CN side. Our observation here is that the MN only *responds* to memory requests but never *initiates* any. This CN-request-MN-respond model allows us to use a custom, connection-less reliable transport protocol that implements almost all transport-layer services and state at CNs, allowing MNs to be free from traditional transport-layer processing. Specifically, our transport protocol manages request IDs, transport logic, retransmission buffer, congestion, and incast control all at CNs. It provides reliability by ordering and retrying an entire memory request at the CN side. As a result, the MN does not need to worry about per-request state or inter-request ordering and only needs a tiny amount of hardware resources which do not grow with the number of clients.

With the above two approaches, the hardware can be largely simplified and thus cheaper, faster, and more scalable. However, we found that ***complete state elimination at MNs is neither feasible nor ideal***. To ensure correctness, the MN has to maintain some state (*e.g.*, to deal with non-idempotent operations). To ensure good data-plane performance, not every operation that involves state should be moved to the low-power SoC or to CNs. Thus, our approach is to eliminate as much state as we can without affecting performance or correctness and to carefully design the remaining state so that it causes small and bounded space and performance overhead.

For example, we perform paging-based virtual-to-physical memory address mapping and access permission checking at the MN hardware pipeline, as these operations are needed for every data access. Page table is a kind of state that could potentially cause

performance and scalability issues but has to be accessed in the data path. We propose a new overflow-free, hash-based page table design where 1) all page table lookups have bounded and low latency (at most one DRAM access time in our implementation), and 2) the total size of all page table entries does not grow with the number of client processes. As a result, even though we cannot eliminate page table from the MN hardware, we can still meet our cost, performance, or scalability requirements.

Another data-plane operation that involves metadata is page fault handling, which is a relatively common operation because we allocate physical memory on demand. Today's page fault handling process is slow and involves metadata for physical memory allocation. We propose a new mechanism to handle page faults in hardware and finish all the handling within bounded hardware cycles. We make page fault handling performance deterministic by moving physical memory allocation operations to software running at the SoC. We further move these allocation operations off the performance-critical path by pre-generating free physical pages to a fix-sized buffer that the hardware pipeline can pull when handling page faults.

We prototyped CBoard with a small set of Xilinx ZCU106 MPSoC FPGA boards [82] 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 [36, 75], a software emulation of hardware-based disaggregated memory [64], and a software-based SmartNIC [48]. 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. Clio is publicly available at https://github.com/WukLab/Clio.

2 GOALS AND RELATED WORKS

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, 19, 72, 78]. With the success of disaggregated storage, researchers in academia and industry have also sought ways to disaggregate memory (and persistent memory) [5, 11, 26, 32, 45, 46, 54, 58, 63–65, 75, 79]. 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 with high utilization. To keep the number of memory devices and total cost of a cluster low, each MN should host hundreds GBs to a few TBs of memory that is expected to be close to fully utilized. To most efficiently use the disaggregated memory, we should allow applications to create and access *disjoint* memory regions of arbitrary sizes at MN.

R2: Supporting a 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 datacenter trends like serverless computing and microservices where applications run as large amounts of small units.

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. Since MemDisagg is network-based, a reasonable performance target of it 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 service-level objectives (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) [16].

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 buggy or malicious clients from reading/writing arbitrary memory at MNs, we should not allow the direct access of MNs' physical memory from the network and MNs should check the access permission.

R6: Low cost. A major goal and benefit of resource disaggregation is cost reduction. A good MemDisagg system should have low *overall* CapEx and OpEx costs. Such a system thus should not 1) use expensive hardware to build MNs, 2) consume huge energy at MNs, and 3) add more costs at CNs than the costs saved at MNs.

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, 63], to offload some application logic to memory devices [63, 66], or to incorporate different network transports [9, 28, 51] and congestion control algorithms [40, 44, 68].

2.2 Server-Based Disaggregated Memory

MemDisagg research so far has mainly taken a server-based approach by using regular servers as MNs [5, 18, 26, 53, 63, 64, 79], usually on top of RDMA. The common limitation of these systems is their reliance on a host server and the resulting CPU energy costs, both of which violate **R6**.

RDMA is what most server-based MemDisagg solutions are based on, with some using RDMA for swapping memory between CNs and MNs [5, 26, 79] and some using RDMA for explicitly accessing MNs [18, 53, 63]. Although RDMA has low average latency and high throughput, it 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

(*PAs*). 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 [18, 76]). Similarly, RNICs cache MR metadata and incur degraded performance when the cache is full. 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 (violating **R4**).

To mitigate the above performance and scalability issues, most RDMA-based systems today [18, 76] preallocate a big MR with huge pages and pin it in physical memory. This results in inefficient memory space utilization and violates **R1**. Even with this approach, there can still be a scalability issue (**R2**), as RDMA needs to create at least one MR for each protection domain (*i.e.*, each client).

In addition to problems caused by RDMA's memory system design, reliable RDMA, the mode used by most MemDisagg solutions, suffers from a connection queue pair (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 [68] and IRN [50] mitigate the above issues by either onloading part of the transport back to software or proposing a new hardware design.

LegoOS [64], our own previous work, 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 to MN PAs. Clio's MN performs the same type of address translation. However, LegoOS emulates MN devices using regular servers and we built 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 inheriting 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 [20, 29, 31, 77]. A recent disaggregated hashing system [86] and our own recent work on disaggregated key-value systems [75] also adopt the PDM model and emulate remote memory with regular servers. To prevent applications from accessing raw physical memory, these solutions add an indirection layer at CNs in hardware [15, 24] or software [75, 86] 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 [75], violating **R3** and **R7**. Second, PDM requires the client side to manage disaggregated memory. For example, CNs need to coordinate with each other or use a global server [75] to perform tasks like memory allocation. Non-MN-side processing is much harder, performs worse compared to memory-side management (violating **R3**), and could even result in higher overall costs because of the high computation added at CNs (violating **R6**). Third, exposing physical memory makes it hard to provide security guarantees (**R5**), as MNs have to authenticate that every access is to a legit physical memory address belonging to the application. Finally, all existing PDM solutions require physical memory pinning at MNs, causing memory wastes and violating **R1**.

```
/* Alloc one remote page. Define a remote lock */
#define PAGE_SIZE (1<<22)
void *remote_addr = ralloc(PAGE_SIZE);
ras_lock lock;

/* Acquire lock to enter critical section.
   Do two AYSNC writes then poll completion. */
void thread1(void *) {
 rlock(lock);
 e[0]=rwrite(remote_addr, local_wbuf1,len, ASYNC);
 e[1]=rwrite(remote_addr+len, local_wbuf2,len, ASYNC);
 runlock(lock);
 rpoll(e, 2);
}

/* Synchronously read from remote */
void thread2(void *) {
 rlock(lock);
 rread(remote_addr, local_rbuf, len, SYNC);
 runlock(lock);
}
```

Figure 1: Example of Using Clio.

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 cause performance overhead that is not always necessary (violating **R3**).

Besides the above PDM works, there are also proposals to include some processing power in between the disaggregated memory layer and the computation layer. soNUMA [55] is a hardware-based solution that scales out NUMA nodes by extending each NUMA node with a hardware unit that services remote memory accesses. Unlike Clio which physically separates MNs from CNs across generic data-center networks, soNUMA still bundles memory and CPU cores, and it is a single-server solution. Thus, soNUMA works only on a limited scale (violating **R2**) and is not flexible (violating **R7**). MIND [42], a concurrent work with Clio, proposes to use a programmable switch for managing coherence directories and memory address mappings between compute nodes and memory nodes. Unlike Clio which adds processing power to every MN, MIND's single programmable switch has limited hardware resources and could be the bottleneck for both performance and scalability.

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 2).

3.1 Clio Interface

Similar to recent MemDisagg proposals [8, 63], our current implementation adopts a non-transparent interface where applications (running at CNs) allocate and access disaggregated memory via explicit API calls. Doing so gives users opportunities to perform application-specific performance optimizations. By design, Clio's APIs can also be called by a runtime like the AIFM runtime [63] or by the kernel/hardware at CN like LegoOS' pComponent [64] to support a transparent interface and allow the use of unmodified user applications. We leave such extension to future work.

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 1 illustrates the usage of Clio with a simple example.

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. `rread` and `rwrite` can then be issued to any allocated VAs. As with the traditional virtual memory interface, allocation and access in RAS are in byte granularity. We offer *synchronous* and *asynchronous* options for `ralloc`, `rfree`, `rread`, and `rwrite`.

Intra-thread request ordering. Within a thread, synchronous APIs follow strict ordering. An application thread that calls a synchronous API blocks until it gets the result. Asynchronous APIs are non-blocking. A calling thread proceeds after calling an asynchronous API and later calls `rpoll` to get the result. Asynchronous APIs follow a release order. Specifically, asynchronous APIs may be executed out of order as long as 1) all asynchronous operations before a `rrelease` complete before the `rrelease` returns, and 2) `rrelease` operations are strictly ordered. On top of this release order, we guarantee that there is no concurrent asynchronous operations with dependencies (Write-After-Read, Read-After-Write, Write-After-Write) and target the same page. The resulting memory consistency level is the same as architecture 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, no read or write to it 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 semantics (*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 a RAS can have their own cached copies at different CNs. Similar to [64], Clio does not make these cached copies coherent automatically and lets applications choose their own coherence protocols. 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.

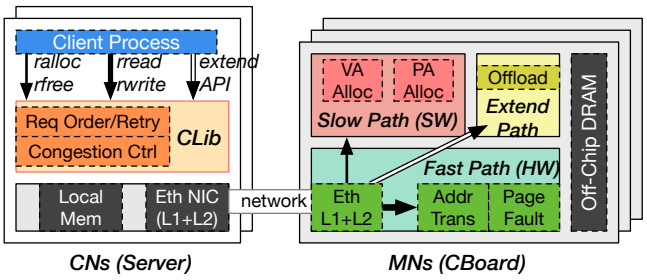

Figure 2: Clio Architecture.

3.2 Clio Architecture

In Clio (Figure 2), CNs are regular servers each equipped with a regular Ethernet NIC and connected to a top-of-rack (ToR) switch. MNs are our customized devices directly connected to a ToR switch. Applications run at CNs on top of our user-space library called *CLib*. It is in charge of request ordering, request retry, congestion, and incast control.

By design, an MN in Clio is a CBoard consisting of an ASIC which runs the hardware logic for all data accesses (we call it the *fast path* and prototyped it with FPGA), an ARM processor which runs software for handling metadata and control operations (*i.e.*, the *slow path*), and an FPGA which 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). Afterward, 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, customized requests with offloaded computation are handled in the extend path.

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 restrictions of hardware 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 fewer resources but suffer from performance loss).

Challenge 2: Data buffers and metadata that the hardware uses should be minimal and have bounded sizes, so that they can be statically planned and fit into the on-chip memory. Unfortunately, traditional software approaches involve various data buffers and metadata that are large and grow with increasing scale. For example, today's reliable network transports maintain per-connection sequence numbers and buffer unacknowledged packets for packet ordering and retransmission, and they grow with the number of connections. Although swapping between on-chip and off-chip memory is possible, doing so would increase both tail latency and hardware logic complexity, especially under large scale.

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 or smooth, and we cannot directly adapt 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 with the following principles that all aim to eliminate state in hardware or bound their performance and space overhead.

Principle 1: Avoid state whenever possible. Not all state in server-based solutions is necessary if we could redesign the hardware. For example, we get rid of RDMA's MR indirection and its metadata altogether by directly mapping application process' RAS VAs to PAs (instead of to MRs then to PAs).

Principle 2: Moving non-critical operations and state 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.

Principle 3: Shifting functionalities and state 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 state and functionalities from MNs to CNs. The key question here is how much and what to shift. 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 state. We store the remaining state 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 amount of physical 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.

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 systems, we use fix-size pages as address allocation and translation unit, while data accesses are in the granularity of byte. Despite the similarity in the goal of address translation, the radix-tree-style, per-address space page table design used by all current architectures [69] does 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 many page table roots, causing additional overhead. Second, a multi-level page table design requires multiple DRAM accesses when there is a translation lookaside buffer (TLB) miss [83]. 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 [52, 74].

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

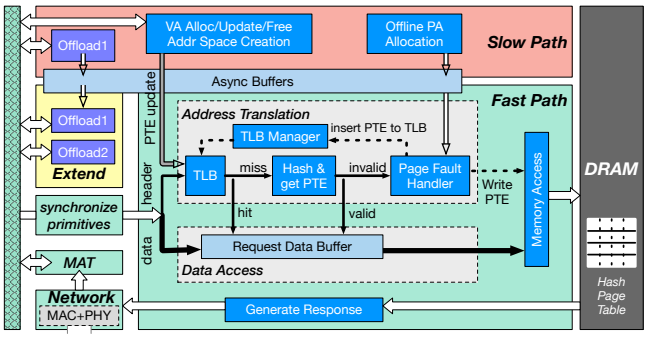

Figure 3: CBoard Design. *Green, yellow, and red areas are anticipated to be built with ASIC, FPGA, and low-power cores.*

is known by the fast path address translation unit, thus avoiding any lookups. 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 the default 4 MB page size, the hash table consumes only 0.4% of the physical memory.

The hash value of a VA and its PID is 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 in a single DRAM access.

A well-known problem with hash-based page table design is hash collisions that could overflow a bucket. Existing hash-based page table designs rely on collision chaining [12] or open addressing [83] 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 allocates VA. The software allocator maintains a per-process VA allocation tree that records allocated VA ranges and permissions, similar to the Linux vma tree [38]. 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 that does not cause hash overflow.

Our design trades potential retry overhead at allocation time (at the slow path) for better run-time performance and simpler hardware design (at the fast path). This overhead is manageable because 1) each retry takes only a few microseconds with our implementation (§5), 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 [80], and 4) we set the page table to have extra slots (2× by default) which absorbs most overflows. 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 into 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 into the page table. Applications that must map at fixed VAs (*e.g.*, libraries) will need to use CN-local memory.

4.3 Low-Tail-Latency Page Fault Handling

A key reason to disaggregate memory is to consolidate memory usages on less DRAM so that memory utilization is higher and the total monetary cost is lower (**R1**). Thus, remote memory space is desired to run close to full capacity, and we allow memory over-commitment at an MN, necessitating page fault handling. Meanwhile, applications like JVM-based ones allocate a large heap memory space at the startup time and then slowly use it to allocate smaller objects [27]. Similarly, many existing far-memory systems [18, 63, 75] allocate a big chunk of remote memory and then use different parts of it for smaller objects to avoid frequently triggering the slow remote allocation operation. In these cases, it is desirable for a MemDisagg system to delay the allocation of physical memory to when the memory is actually used (*i.e.*, *on-demand* allocation) or to "reshape" memory [67] during runtime, necessitating page fault handling.

Page faults are traditionally signaled by the hardware and handled by the OS. This is a slow process because of the costly interrupt and kernel-trapping flow. For example, a remote page fault via RDMA costs 16.8 *ms* from our experiments using Mellanox ConnectX-4. To avoid page faults, most RDMA-based systems 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, violating **R1** and **R2**.

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. Handling initial-access faults in hardware is challenging, as initial accesses require PA allocation, which is a slow operation that involves manipulating complex data structures. Thus, we handle PA allocation in 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 slow down the data plane.

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 requests to proceed immediately.

A recent work [41] also handles page faults 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 faults in hardware with the goal of low tail latency, and we built it in FPGA.

Putting the 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 Asymmetric Network Tailored for MemDisagg

With large 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 fits 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 state, *e.g.*, per-connection sequence numbers, congestion state [9], and bitmaps [47, 50], 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 (except for memory migration from one MN to another MN (§4.7), in which case we use another simple protocol to achieve the similar goal). Moreover, these requests are all memory-related operations that have their specific properties. With these insights, we design a new network system with two main ideas. Our first idea is to maintain transport logic, state, and data buffers only at CNs, essentially making MNs "transportless" (**Principle 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 an Ethernet NIC. CNs use regular, commodity Ethernet NICs and regular Ethernet switches to connect to MNs. MNs include only standard Ethernet physical, link, and network layers and a slim layer for handling corner-case requests (§4.5). We now describe our detailed design.

Removing connections with request-response semantics. Connections (*i.e.*, QPs) are a major scalability issue with RDMA. Similar to recent works [51, 68], we make our network system connection-less using request-response pairs. Applications running at CNs directly initiate Clio APIs to an MN without any connections. CLib assigns a unique request ID to each request. The MN attaches the same request ID when sending the response back. CLib uses responses as ACKs and matches a response with an outstanding request using the request ID. Neither CNs nor MNs send ACKs.

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 in the sending or the receiving direction. On the receiving path, MN's network stack only checks a packet's integrity at the link layer. 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 happens: a NACK is received, the response from MN is corrupted, or no response is received within a `TIMEOUT` period. 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 minimize state at MN. To this end, we build the entire congestion and incast control at the CN in the CLib. To control congestion, CLib adopts 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 [40, 49, 68]. 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) algorithm.

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*, which controls the maximum bytes of expected responses. CLib sends a request only when both *cwnd* and *iwnd* have room.

Handling incast to an MN is more challenging, as we cannot throttle incoming traffic at the MN side or would otherwise maintain state at MNs. To have CNs handle incast to MNs, we draw inspiration from Swift [40] 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. Essentially, this situation happens when the network between a CN and an MN is really congested, and the only way is to slow the sending speed.

4.5 Request Ordering and Data Consistency

As explained in §3.1, Clio supports both synchronous and asynchronous remote memory APIs, with the former following a sequential, one-at-a-time order in a thread and the latter following

a release order in a thread. Furthermore, Clio provides synchronization primitives for inter-thread consistency. We now discuss how Clio achieves these correctness guarantees by presenting our mechanisms for handling intra-request intra-thread ordering, inter-request intra-thread ordering, inter-thread consistency, and retries. At the end, we will provide the rationales behind our design.

One difficulty in designing the request ordering and consistency mechanisms is our relaxed network ordering guarantees, which we adopt to minimize the hardware resource consumption for the network layer at MNs (§4.4). On an asynchronous network, it is generally hard to guarantee any type of request ordering when there can be multiple outstanding requests (either multiple threads accessing shared memory or a single thread issuing multiple asynchronous APIs). It is even harder for Clio because we aim to make MN stateless as much as possible. Our general approaches are 1) using CNs to ensure that no two concurrently outstanding requests are dependent on each other, and 2) using MNs to ensure that every user request is only executed once even in the event of retries.

Allowing intra-request packet re-ordering (T1). A request or a response in Clio can contain multiple link-layer packets. Enforcing packet ordering above the link layer normally requires maintaining state (*e.g.*, packet sequence ID) at both the sender and the receiver. To avoid maintaining such state 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 link-layer maximum transmission unit (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 [50]. 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. When a CN receives multiple link-layer packets belonging to the same request response, CLib reassembles them before delivering them to the application.

Enforcing intra-thread inter-request ordering at CN (T2). Since only one synchronous request can be outstanding in a thread, there cannot be any inter-request reordering problem. On the other hand, there can be multiple outstanding asynchronous requests. Our provided consistency level disallows concurrent asynchronous requests that are dependent on each other (WAW, RAW, or WAR). In addition, all requests must complete 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.

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 downside 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 (T3). 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. Synchronization primitives are one of the only two cases where MN needs to maintain state. As these operations are infrequent and each of these operations executes in bounded time, the hardware resources for maintaining their state are minimal and bounded.

Handling retries (T4). CLib retries a request after a `TIMEOUT` period without receiving any response. Potential consistency problems could happen as CBoard could execute a retried write after the data is written by another write request thus undoing this other request's write. Such situations could happen when the original request's response is lost or delayed and/or when the network reorders packets. We use two techniques to solve this problem.

First, CLib attaches a new request ID to each retry, essentially making it a new request with its own matching response. Together with CLib's ordering enforcement, it ensures that there is only one outstanding request (or a retry) at any time. Second, we maintain a small buffer at MN to record the request IDs of recently executed writes and atomic APIs and the results of the atomic APIs. A retry attaches its own request ID and the ID of the failed request. If MN finds a match of the latter in the buffer, it will not execute the request. For atomic APIs, it sends the cached result as the response. We set this buffer's size to be $3 \times \texttt{TIMEOUT} \times \textit{bandwidth}$, which is 30 KB in our setting. It is one of the only two types of state MN maintains and does not affect the scalability of MN, since its size is statically associated with the link bandwidth and the `TIMEOUT` value. With this size, the MN can "remember" an operation long enough for two retries from the CN. Only when both retries and the original request all fail, the MN will fail to properly handle a future retry. This case is extremely rare [51], and we report the error to the application, similar to [36, 68].

Why T1 to T4? We now briefly discuss the rationale behind why we need all T1 to T4 to properly deliver our consistency guarantees. First, assume that there is no packet loss or corruption (*i.e.*, no retry) but the network can reorder packets. In this case, using T1 and T2 alone is enough to guarantee the proper ordering of Clio memory operations, since they guarantee that network reordering will only affect either packets within the same request or requests that are not dependent on each other. T3 guarantees the correctness of synchronization primitives since the MN is the serialization point and is where these primitives are executed. Now, consider the case where there are retries. Because of the asynchronous network, a timed-out request could just be slow and still reach the MN, either before or after the execution of the retried request. If

another request is executed in between the original and the retried requests, inconsistency could happen (*e.g.*, losing the data of this other request if it is a write). The root cause of this problem is that one request can be executed twice when it is retried. T4 solves this problem by ensuring that the MN only executes a request once even if it is retried.

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 application computation offloading in the extend path. Users can write and deploy application offloads both in FPGA and in software (run in the ARM). 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 PID and virtual memory address space, and they use the same virtual memory APIs (§3.1) to access on-board memory. It could also share data with processes running at CNs in the same way that two CN processes share memory. Finally, an offload's data and control paths could be split to FPGA and ARM and use the same async-buffer mechanism for communication between them. These unique designs made developing computation offloads easier and closer to traditional multi-threaded software programming.

4.7 Distributed MNs

Our discussion so far focused on a single MN (CBoard). To more efficiently use remote memory space and to allow one application to use more memory than what one CBoard can offer, we extend the single-MN design to a distributed one with multiple MNs. Specifically, an application process' RAS can span multiple MNs, and one MN can host multiple RASs. We adopt LegoOS' two-level distributed virtual memory management approach to manage distributed MNs in Clio. A global controller manages RASs in coarse granularity (assigning 1 GB virtual memory regions to different MNs). Each MN then manages the assigned regions at fine granularity.

The main difference between LegoOS and Clio's distributed memory system is that in Clio, each MN can be over-committed (*i.e.*, allocating more virtual memory than its physical memory size), and when an MN is under memory pressure, it migrates data to another MN that is less pressured (coordinated by the global controller). The traditional way of providing memory over-commitment is through memory swapping, which could be potentially implemented by swapping memory between MNs. However, swapping would cause performance impact on the data path and add complexity to the hardware implementation. Instead of swapping, we *proactively* migrate a rarely accessed memory region to another MN when an MN is under memory pressure (its free physical memory space is below a threshold). During migration, we pause all client requests to the region being migrated. With our 10 Gbps experimental board, migrating a 1 GB region takes 1.3 second. Migration happens rarely and, unlike swapping, happens in the background. Thus, it has little disturbance to foreground application performance.

5 CLIO IMPLEMENTATION

Apart from challenges discussed in §4, our implementation of Clio also needs to overcome several practical challenges, for example, how can different hardware components most efficiently work together in CBoard, how to minimize software overhead in CLib. This section describes how we implemented CBoard and CLib, focusing on the new techniques we designed to overcome these challenges. Currently, Clio consists of 24.6K SLOC (excluding computation offloads and third-party IPs). They include 5.6K SLOC in SpinalHDL [70] and 2K in C HLS for FPGA hardware, and 17K in C for CLib and ARM software. We use vendor-supplied interconnect and DDR IPs, and an open-source MAC and PHY network stack [21].

CBoard Prototyping. We prototyped CBoard with a low-cost ($2495 retail price) Xilinx MPSoC board [82] and build the hardware fast path (which is anticipated to be built in ASIC) with FPGA. 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 bandwidth). Achieving II of one is not easy and requires careful pipeline design in all the modules. With II one, our data path can achieve a maximum of 128 Gbps throughput even with just the slower FPGA clock frequency and would be higher with real ASIC implementation.

Our prototyping 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 target, 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) and 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 latency. With this board, the ARM's access to on-board DRAM is much slower than the FPGA's access because the ARM 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.

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, so that the control path can read page table content faster. When the control path needs to perform a virtual memory space allocation, it reads the shadow page table to test if an address would cause an overflow (§4.2). We keep the shadow page table in sync with the real page table by updating both tables when adding, removing, or updating the page table entries.

In addition to maintaining shadow metadata, 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 fewer 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, 44, 50, 85], we rely on our congestion and incast control to avoid triggering PFC as much as possible.

Finally, 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.

CLib Implementation. Even though we optimize the performance of CBoard, the end-to-end application performance can still be hugely impacted if the host software component (CLib) is not as fast. Thus, our CLib implementation aims to provide low-latency performance by adopting several ideas (e.g., data inlining, doorbell batching) from recent low-latency I/O solutions [34–37, 57, 76, 84]. We implemented CLib in the user space. It has three parts: a user-facing request ordering layer that performs dependency check and ordering of address-conflicting requests, a transport layer that performs congestion/incast control and request-level retransmission, and a low-level device driver layer that interacts with the NIC (similar to DPDK [17] but simpler). CLib bypasses kernel and directly issues raw Ethernet requests to the NIC with zero memory copy. For synchronous APIs, we let the requesting thread poll the NIC for receiving the response right after each request. For asynchronous APIs, the application thread proceeds with other computations after issuing the request and only busy polls when the program calls `rpoll`.

6 BUILDING APPLICATIONS ON CLIO

We built five applications on top of Clio, one that uses the basic Clio APIs, one that implements and uses a high-level, extended API, and two that offload data processing tasks to MNs, and one that splits computation across CNs and 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 MNs, one for compressed and one for original, both allocated with `ralloc`. 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`. Note that we use compression and decompression as an example of image processing. These operations could potentially be offloaded to MNs. However, in reality, there can be many other types of image processing that are more complex and are hard and costly to implement in hardware, necessitating software processing at CNs. 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 [63] could follow this example to make simple changes in their libraries to run on Clio. We first built an extended pointer-chasing functionality in FPGA at the MN which follows pointers in a linked list and performs a value comparison at each traversed list node. It returns either the node value when there is a match or null when the next pointer becomes null. We then expose this functionality to CNs as an extended API. The software running at CN allocates a big contiguous remote memory space using `ralloc` and uses this space to store radix tree nodes. Nodes in each layer are linked to a list. To search a radix tree, the CN software goes through each layer of the tree and calls the pointer chasing API until a match is found. We implemented the radix tree with 300 C code at CN and 150 SpinalHDL code at CBoard 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. Clio-KV runs at an MN as a computation offloading module. Users can access it through a key-value interface from multiple CNs. The Clio-KV module has its own virtual memory address space and uses Clio virtual memory APIs to access it. Clio-KV uses a chained hash table in its virtual memory space for managing the metadata of key-value pairs, and it stores the actual key values at separate locations in the space. Each hash bucket has a chain of slots. Each slot contains the virtual addresses of seven key-value pairs. It also stores a fingerprint for each key-value pair.

To create a new key-value pair, Clio-KV allocates space for the key-value data with an `ralloc` call and writes the data with an `rwrite`. It then calculates the hash and the fingerprint of the key. Afterward, it fetches the last hash slot in the corresponding hash bucket using the hash value. If that slot is full, Clio-KV allocates another slot using `ralloc`; otherwise, it just uses the fetched last slot. It then inserts the virtual address and fingerprint of the data into the last/new slot. Finally, it links the current last slot to the new slot if a new one is created.

To perform a read, Clio-KV locates the hash bucket (with the key's hash value) and fetches one slot in the bucket chain at a time using `rread`. It then compares the fingerprint of the key to the seven entries in the slot. If there is no match, it fetches the next slot in the bucket. Otherwise, with a matched entry, it reads the key-value pair using the address stored in that entry with an `rread`. It then compares the full key and returns the value if it is a match. Otherwise, it keeps searching the bucket.

The above describes a single-MN Clio-KV system. Another CN-side load balancer is used to partition key-value pairs into different MNs. Since all CNs requests of the same partition go to the same MN and Clio APIs within an MN are properly ordered, it is fairly easy for Clio-KV to guarantee the atomic-write, read-committed consistency level.

We implemented Clio-KV with 772 SpinalHDL code in 6 developer days. To evaluate Clio's virtual memory API overhead at CBoard, we also implemented a key-value store with the same design as Clio-KV but with raw physical memory interface. This physical-memory-based implementation takes more time to develop and only yields 4%–12% latency improvement and 1%–5% throughput improvement over Clio-KV.

Multi-version object store. We built a multi-version object store (*Clio-MV*) which lets users on CNs create an object, append a new version to an object, read a specific version or the latest version of an object, and delete an object. Similar to Clio-KV, Clio-MV has its own address space. In the address space, it uses an array to store

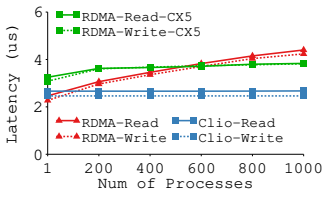

Figure 4: Process (Connection) Scalability.

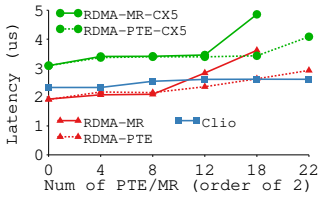

Figure 5: PTE and MR Scalability. *RDMA fails beyond 2^{18} MRs.*

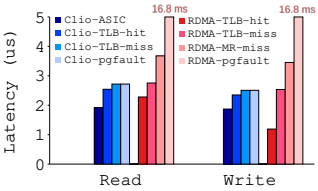

Figure 6: Comparison of TLB Miss and page fault. *Clio-ASIC are projected values of TLB hit.*

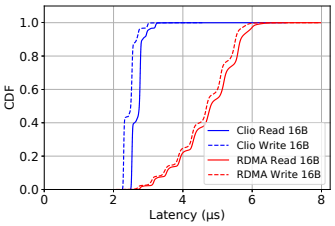

Figure 7: Latency CDF.

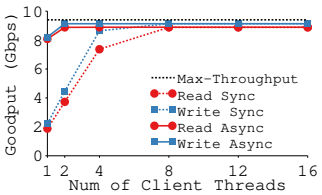

Figure 8: End-to-End Goodput. *1 KB requests.*

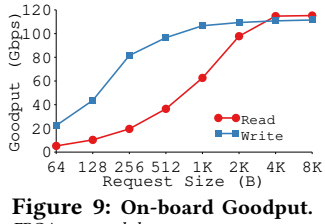

Figure 9: On-board Goodput. *FPGA test module generates requests at maximum speed.*

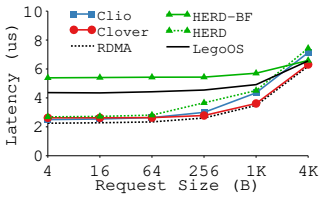

Figure 10: Read Latency. *HERD-BF: HERD running on BlueField.*

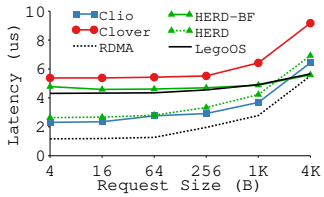

Figure 11: Write Latency. *Clover requires ≥ 2 RTTs for write.*

versions of data for each object, a map to store the mapping from object IDs to the per-object array addresses, and a list to store free object IDs. When a new object is created, Clio-MV allocates a new array (with `ralloc`) and writes the virtual memory address of the array into the object ID map. Appending a new version to an object simply increases the latest version number and uses that as an index to the object array for writing the value. Reading a version simply reads the corresponding element of the array.

Clio-MV allows concurrent accesses from CNs to an object and guarantees sequential consistency for each object. Each Clio-MV user request involves at least two internal Clio operations, some of which include both metadata and data operations. This compound request pattern makes it tricky to deal with synchronization problems, as Clio-MV needs to ensure that no internal Clio operation of a later Clio-MV request could affect the correctness of an earlier Clio-MV request. We implemented Clio-MV with 1680 lines of C HLS code in 15 developer days.

Simple data analytics. Our final example is a simple DataFrame-like data processing application (*Clio-DF*), which splits its computation between CN and MN. We implement `select` and `aggregate` at MN as two offloads, as offloading them can reduce the amount of data sent over the network. We keep other operations like `shuffle` and `histogram` at CN. For the same user, all these modules share the same address space regardless of whether they are at CN or MN. Thanks to Clio's support of computation offloading sharing the same address space as computations running at host, Clio-DF's implementation is largely simplified and its performance is improved by avoiding data serialization/deserialization. We implemented Clio-DF with 202 lines of SpinalHDL code and 170 lines of C interface code in 7 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.

Environment. We evaluated Clio in our local cluster of four CNs and four MNs (Xilinx ZCU106 boards), all connected to an 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 ConnectX-3 NIC, with two of them also having an Nvidia BlueField SmartNIC [48]. We also include results from CloudLab [14] with the Nvidia ConnectX-5 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 persists with newer generations of RNIC, which is also confirmed by our previous works [56, 74].

Figure 5 evaluates the scalability with respect to PTEs and memory regions. For the memory region test, we register multiple MRs using the same physical memory for RDMA. For Clio, we map a large range of VAs (up to 4 TB) to a small physical memory space, as our testbed only has 2 GB physical memory. However, the number of PTEs and the amount of processing needed are the same for CBoard as if it had a real 4 TB physical memory. Thus, this workload stress tests CBoard's scalability. 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). It has two levels of latency that are both stable: a lower latency below 2^4 for TLB hit and a higher latency above 2^4 for TLB miss (which always involves one DRAM access). A 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, when the

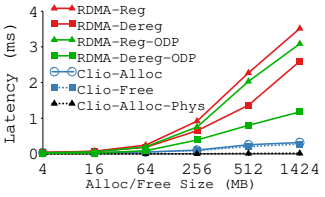

Figure 12: Alloc/Free Latency.
ODP means On-Demand-Paging mode

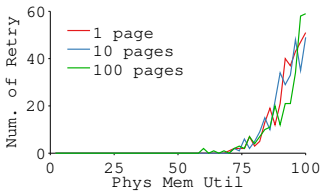

Figure 13: Alloc Retry Rate.

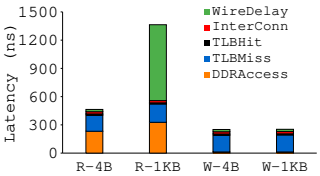

Figure 14: Latency Breakdown.
Breakdown of time spent at CBoard.

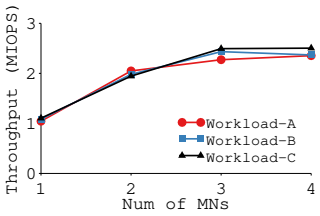

Figure 15: Clio-KV Scalability against MNs.

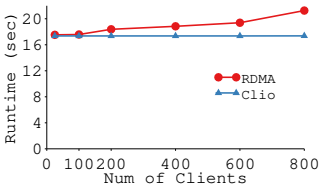

Figure 16: Image Compression.

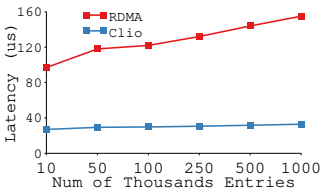

Figure 17: Radix Tree Search Latency.

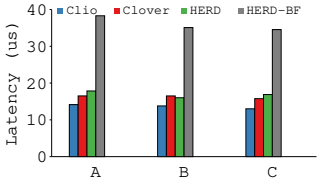

Figure 18: Key-Value Store YCSB Latency.

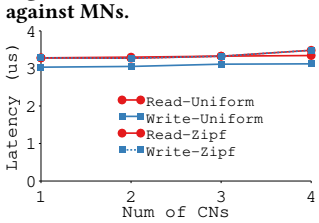

Figure 19: Clio-MV Object Read/Write Latency.

MR metadata is not in RNIC). RDMA's performance degrades significantly with misses. Its page fault handling is extremely slow (16.8 *ms*). We confirm the same effect on CloudLab with the newer ConnectX-5 NICs. 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, time spent on third-party FPGA IPs, number of cycles on FPGA, and time on accessing on-board DRAM. We maintain the first two parts, 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). This estimation is conservative, as a real ASIC implementation of the third-party IPs would make the total latency lower. Our estimated read latency is better than RDMA, while write latency is worse. We suspect the reason being Nvidia RNIC's optimization of replying a write before it is fully written to DRAM, which Clio could also potentially adopt.

Figure 7 plots the request latency CDF of continuously running read/write 16 B data while not triggering page faults. Even without page faults, 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 of Clio's FPGA implementation without the bottleneck of the board's 10 Gbps port, by generating traffic on board. 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 found the throughput bottleneck to be the third-party non-pipelined DMA IP (which could potentially be improved).

Comparison with other systems. We compare Clio with native one-sided RDMA, Clover [75], HERD [36], and LegoOS [64]. We ran HERD on both CPU and BlueField (HERD-BF). Clover is a passive disaggregated persistent memory system which 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 builds its virtual memory system in software at MN.

Clio's performance is similar to HERD and close to native RDMA. Clover's write is the worst because 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 ConnectX-5 chip and ARM processor chip. LegoOS's latency is almost two 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 its software approach, demonstrating the necessity of a hardware-based solution like Clio.

Allocation performance. Figure 12 shows Clio's VA and PA allocation and RDMA's MR registration performance. 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 allocation sizes as the physical memory fills 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 the latencies.

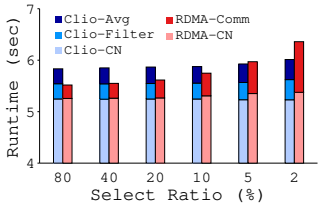

Figure 20: Select-Aggregate-Shuffle. *Y axis starts at 4 sec. CN represents computation done at CN.*

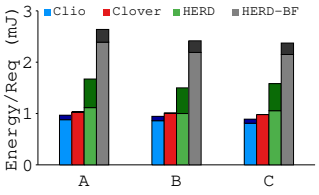

Figure 21: Energy Comparison. *Darker/lighter shades represent energy spent at MNs and CNs.*

System/Module	Logic (LUT)	Memory (BRAM)
StRoM-RoCEv2	39%	76%
Tonic-SACK	48%	40%
Clio (Total)	31%	31%
VirtMem	5.5%	3%
NetStack	2.3%	1.7%
Go-Back-N	5.8%	2.6%

Figure 22: FPGA Utilization.

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 16 shows the total runtime per client. We compare Clio with RDMA, with both performing computation at the CN side and the RDMA using one-sided operations instead of Clio APIs to read/write images in remote memory. 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 to have protected memory accesses. With more MRs, RDMA runs into the case where the RNIC cannot hold all the MR metadata and many accesses would involve a slow read to host main memory.

Radix Tree. Figure 17 shows the latency of searching a key in pre-populated radix trees when varying the tree size. We again compare with RDMA which uses one-sided read operations to perform the tree traversal task. 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 (each tree level). In addition, RDMA also scales worse than Clio.

Key-value store. Figure 18 evaluates Clio-KV using the YCSB benchmark [1] and compares 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 (θ = 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 performs the best. HERD running on BlueField performs the worst, mainly because BlueField's slower crossing between its NIC chip and ARM chip.

Figures 15 shows the throughput of Clio-KV when varying the number of MNs. Similar to our Clio scalability results, Clio-KV can reach a CN's maximum throughput and can handle concurrent get/set requests even under contention. These results are similar to or better than previous FPGA-based and RDMA-based key-value stores that are fine-tuned for just key-value workloads (Table 3 in [43]), while we got our results without any performance tuning.

Multi-version data store. We evaluate Clio-MV by varying the number of CNs that concurrently access data objects (of 16 B) on an MN using workloads of 50% read (of different versions) and 50% write under uniform and Zipf distribution of objects (Figure 19). Clio-MV's read and write have the same performance, and reading any version has the same performance, since we use an array-based version design.

Data analytics. We run a simple workload which first `selects` rows in a table whose field-A matches a value (*e.g.*, gender is female) and then calculates avg of field-B (*e.g.*, final score) of all the rows. Finally, it calculates the histogram of the selected rows (*e.g.*, score distribution), which can be presented to the user together with the avg value. Clio executes the first two steps at MN offloads and the final step at CN, while RDMA always reads rows to CN and then does each operation. Figure 20 plots the total run time as the select ratio decreases (*i.e.*, fewer rows selected). When the select ratio is low, Clio transfers much less data than RDMA, resulting in its better performance.

7.3 CapEx, Energy, and FPGA Utilization

We estimate the cost of server and CBoard using market prices of different hardware units. When using 1 TB DRAM, a server-based MN costs 1.1-1.5× and consumes 1.9-2.7× power compared to CBoard. These numbers become 1.4-2.5× and 5.1-8.6× with OptaneDimm [60], which we expect to be the more likely remote memory media in future systems.

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 [59], and FPGA (measured). We omit the energy used by DRAM and NICs in all the calculations. 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 of 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 22 compares the FPGA utilization among Clio, StRoM's RoCEv2 [66], and Tonic's selective ack stack [9]. Both StRoM and Tonic include only a network stack but 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). Overall, our efficient hardware implementation leaves most FPGA resources available for application offloads.

8 DISCUSSION AND CONCLUSION

We presented Clio, a new hardware-based disaggregated memory platform. Our FPGA prototype demonstrates that Clio achieves great performance, scalability, and cost-saving. This work not only guides the future development of MemDisagg solutions but also demonstrates how to implement a core OS subsystem in hardware and co-design it with the network. We now present our concluding thoughts with several open questions.

Security and performance isolation. Clio's protection domain is a user process, which is the same as the traditional single-server process-address-space-based protection. The difference is that Clio performs permission checks at MNs: it restricts a process' access to only its (remote) memory address space and does this check based on the global PID. Thus, the safety of Clio relies on PIDs to be authentic (*e.g.*, by letting a trusted CN OS or trusted CN hardware attach process IDs to each Clio request). There have been researches on attacking RDMA systems by forging requests [62] and on adding security features to RDMA [68, 71]. How these and other existing security works relate and could be extended in a memory disaggregation setting is an open problem, and we leave this for future work.

There are also designs in our current implementation that could be improved to provide more protection against side-channel and DoS attacks. For example, currently, the TLB is shared across application processes, and there is no network bandwidth limit for an individual connection. Adding more isolation to these components would potentially increase the cost of CBoard or reduce its performance. We leave exploring such tradeoffs to future work.

Failure handling. Although memory systems are usually assumed to be volatile, there are still situations that require proper failure handling (*e.g.*, for high availability or to use memory for storing data). As there can be many ways to build memory services on Clio and many such services are already or would benefit from handling failure on their own, we choose not to have any built-in failure handling mechanism in Clio. Instead, Clio should offer primitives like replicated writes for users to build their own services. We leave adding such API extensions to Clio as future work.

CN-side stack. An interesting finding we have is that CN-side systems could become a performance bottleneck after we made the remote memory layer very fast. Surprisingly, most of our performance tuning efforts are spent on the CN side (*e.g.*, thread model, network stack implementation). Nonetheless, software implementation is inevitably slower than customized hardware implementation. Future works could potentially improve Clio's CN side performance by offloading the software stack to a customized hardware NIC.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers and our shepherd Mark Silberstein for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We are also thankful to Geoff Voelker, Harry Xu, Steven Swanson, Alex Forencich for their valuable feedback on our work.

This material is based upon work supported by the National Science Foundation under the following grant: NSF 2022675, and gifts from VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

REFERENCES

[1] [n.d.]. YCSB Github Repository. https://github.com/brianfrankcooper/YCSB.
[2] Intel Xeon Gold 5128. [n.d.]. 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. 2019. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. Bertinoro, Italy.
[4] Alibaba. [n.d.]. "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. [n.d.]. Can Far Memory Improve Job Throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.
[6] Amazon. 2019. Amazon Elastic Block Store. https://aws.amazon.com/ebs/?nc1=h_ls.
[7] Amazon. 2019. Amazon S3. https://aws.amazon.com/s3/.
[8] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. [n.d.]. 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. [n.d.]. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*.
[10] ARMv8. [n.d.]. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-a-architecture-2016-additions.
[11] Krste Asanović. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. Keynote talk at the 12th USENIX Conference on File and Storage Technologies (FAST '14).
[12] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*.
[13] Brian Cho and Ergin Seyfe. 2019. Taking Advantage of a Disaggregated Storage and Compute Architecture. In *Spark+AI Summit 2019 (SAIS '19)*. San Francisco, CA, USA.
[14] CloudLab. [n.d.]. https://www.cloudlab.us/.
[15] CXL Consortium. [n.d.]. https://www.computeexpresslink.org/.
[16] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext
[17] DPDK. [n.d.]. https://www.dpdk.org/.
[18] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*. Seattle, WA, USA.
[19] Facebook. 2017. Introducing Bryce Canyon: Our next-generation storage platform. https://code.fb.com/data-center-engineering/introducing-bryce-canyon-our-next-generation-storage-platform/.
[20] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. 2015. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS '15)*. Kartause Ittingen, Switzerland.
[21] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. 2020. Corundum: An Open-Source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '20)*. Fayetteville,AK.
[22] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.
[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. 2021. When Cloud Storage Meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*.
[24] Gen-Z Consortium. [n.d.]. https://genzconsortium.org.
[25] Albert Greenberg, Gisli Hjalmtysson, Dave Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. 2005. A Clean Slate 4D Approach to Network Control and Management ^—. *ACM SIGCOMM Computer Communication Review* (October 2005).
[26] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*. Boston, MA, USA.
[27] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*. IEEE, 1–10.
[28] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. [n.d.]. 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)*.
[29] Hewlett Packard. 2005. The Machine: A New Kind of Computer. http://www.hpl.hp.com/research/systems-research/themachine/.
[30] Hewlett-Packard. 2010. Memory Technology Evolution: An Overview of System Memory Technologies the 9th edition. https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-c00256987.

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

[32] Intel Corporation. [n.d.]. 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.

[33] ITRS. [n.d.]. International Technology Roadmap for Semiconductors (SIA) 2014 Edition.

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

[35] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*. Boston, MA, USA.

[36] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. 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.

[37] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC '16)*. Denver, CO, USA.

[38] Linux Kernel. [n.d.]. Red-black Trees (rbtree) in Linux. https://www.kernel.org/doc/Documentation/rbtree.txt.

[39] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. 2014. Network Virtualization in Multi-tenant Datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA.

[40] 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. 2020. 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)*.

[41] Gyusun Lee, Wenjing Jin, Wonsuk Song, Jeonghun Gong, Jonghyun Bae, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. 2020. A Case for Hardware-Based Demand Paging. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA '20)*.

[42] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 488–504.

[43] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Shanghai, China.

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

[45] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. Austin, Texas.

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

[47] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2017. Memory Efficient Loss Recovery for Hardware-Based Transport in Datacenter. In *Proceedings of the First Asia-Pacific Workshop on Networking (APNet'17)*.

[48] Mellanox. 2018. BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[49] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. [n.d.]. TIMELY: RTT-based Congestion Control for the Datacenter. *ACM SIGCOMM Computer Communication Review (SIGCOMM '15)* ([n. d.]).

[50] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*.

[51] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. [n.d.]. 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)*.

[52] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*.

[53] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. 2018. Welcome to Zombieland: Practical and Energy-Efficient Memory Disaggregation in a Datacenter. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*.

[54] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. 2018. Welcome to Zombieland: Practical and Energy-efficient Memory Disaggregation in a Datacenter. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Porto, Portugal.

[55] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. *ACM SIGPLAN Notices* 49, 4 (2014), 3–18.

[56] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. [n.d.]. Storm: A Fast Transactional Dataplane for Remote Data Structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR '19)*.

[57] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[58] 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. 2015. The RAMCloud Storage System. *ACM Transactions Computer System* 33, 3 (August 2015), 7:1–7:55.

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

[60] Intel Optane persistent memory. [n.d.]. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html.

[61] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)* 33, 4 (2015), 1–30.

[62] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. 2021. ReDMArk: Bypassing RDMA Security Mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)*.

[63] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. [n.d.]. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.

[64] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. Carlsbad, CA.

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

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

[67] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. 2021. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*.

[68] 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. [n.d.]. 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)*.

[69] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. 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)*.

[70] SpinalHDL. [n.d.]. SpinalHDL. https://github.com/SpinalHDL/SpinalHDL.

[71] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. 2020. sRDMA – Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*.

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

[73] TECHPP. 2019. 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/.

[74] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. [n.d.]. Pythia: Remote Oracles for the Masses. In *28th USENIX Security Symposium (USENIX Security 19)*.

[75] Shin-Yeh Tsai, Yizhou Shan, , and Yiying Zhang. 2020. 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.

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

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

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

[79] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. [n.d.]. Semeru: A Memory-Disaggregated Managed Runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.

[80] Wikipedia. [n.d.]. "Jenkins hash function". https://en.wikipedia.org/wiki/Jenkins_hash_function.

[81] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News* 23, 1 (March 1995).

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

[83] Idan Yaniv and Dan Tsafrir. 2016. 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)*.

[84] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*.

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

[86] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*.