Cowbird: Freeing CPUs to Compute by Offloading the Disaggregation of Memory

Xinyi Chen
University of Pennsylvania
cxinyic@seas.upenn.edu

Liangcheng Yu
University of Pennsylvania
leoyu@seas.upenn.edu

Vincent Liu
University of Pennsylvania
liuv@seas.upenn.edu

Qizhen Zhang
U of T and Microsoft
qz@cs.toronto.edu

ABSTRACT

Memory disaggregation allows applications running on compute servers to expand their pool of available memory capacity by leveraging remote resources through low-latency networks. Unfortunately, in existing software-level disaggregation frameworks, the simple act of issuing requests to remote memory—paid on every access—can consume many CPU cycles. This overhead represents a direct cost to disaggregation, not only on the throughput of remote memory access but also on application logic, which must contend with the framework's CPU overheads.

In this paper, we present Cowbird, a memory disaggregation architecture that frees compute servers to fulfill their stated purpose by removing disaggregation-related logic from their CPUs. Our experimental evaluation shows that Cowbird eliminates disaggregation overhead on compute-server CPUs and can improve end-to-end application performance by up to 3.5× compared to RDMA-only communication.

CCS CONCEPTS

• **Networks** → *Programmable networks*; **In-network processing**; • **Software and its engineering** → **Software system structures**.

KEYWORDS

Memory disaggregation, RDMA, Compute offload, Programmable networks, Spot VMs, P4 programmable switches, SmartNICs

ACM Reference Format:
Xinyi Chen, Liangcheng Yu, Vincent Liu, and Qizhen Zhang. 2023. Cowbird: Freeing CPUs to Compute by Offloading the Disaggregation of Memory. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3603269.3604833

1 INTRODUCTION

Memory disaggregation—a design methodology in which applications are allowed to leverage physically distinct pools of memory—promises data centers substantial operational benefits [4, 8, 11, 17, 20, 24, 32, 35, 40, 47]. To operators, it provides better cost efficiency, increased flexibility when provisioning/upgrading hardware, and reduced resource fragmentation compared to monolithic designs. To users, it provides significantly increased memory elasticity.

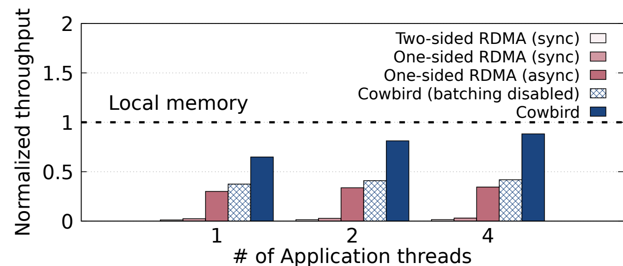


Figure 1: Throughput of hash index probe of 256-byte elements in remote memory using different communication primitives, normalized to the performance of local memory. Just calling an RDMA function induces large application overheads; in comparison, Cowbird bridges the gap between the performance of local and remote memory.

Disaggregation can take many forms, but broadly speaking, recent proposals fall into one of two categories. The first is hardware-level disaggregation like that envisioned by the CXL standard [2], which leverages processor support and a specialized CXL interconnect to hide disaggregation behind traditional load/store instructions and a NUMA-like abstraction. The second is the class of architectures that integrate disaggregation into the application or OS layers, e.g., through RDMA verbs to far memory. Compared to hardware-level solutions, these approaches enable access to a wider set of memory resources, the possibility of application-specific optimizations (e.g., to cache coherency), and crucially, are available and deployable today.

Unfortunately, without CPU-architecture support, it can be difficult to make existing software-level disaggregation mechanisms performant. For example, in OS-level memory disaggregation (e.g., LegoOS [35], Infiniswap [17], and Fastswap [5]), compute nodes must execute blocking page faults that switch execution contexts, dispatch RPCs/RMAs to the remote memory server, and poll for completions when the remote data is finally paged into the local cache. Even application-level mechanisms that leverage kernel-bypass and lightweight cooperative multitasking (e.g., Redy [47] and AIFM [32]) must still consume CPU cycles/cores to issue RPCs and otherwise manage communication.

As one example of this effect, consider the fetch of a piece of data from remote memory using RDMA, which is already significantly more efficient than the TCP/socket interface. Most RDMA-based disaggregation frameworks implement these read RPC calls using two-sided or double one-sided (one from the compute node and one from the memory pool) RDMA operations [14, 35, 39, 47]. In these RPCs, the client posts a request for data and busy polls until the memory pool receives the request, issues one of its own to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604833

write the data to the client memory, and the data finishes writing. Waiting for this sequence to complete takes orders of magnitude longer than local memory accesses and can substantially reduce application performance. Figure 1 quantifies this slowdown using a micro-benchmark of a hash index probe of remote data in the testbed described in Section 7.

Many systems have tried to optimize the above process. For example, eliminating the memory-pool logic can reduce the critical execution path to include just a single one-sided operation from the client; however, as shown in Figure 1, the performance benefits of this optimization are minimal. Similarly, recent work has tried to overlap communication and computation by separating the posting of RDMA requests and polling for their completions, progressing on other tasks in the interim using application-level cooperative multitasking mechanisms [32, 47]. In the end, however, even the simple act of calling two local RDMA functions, a post followed by a later poll, can result in substantial overheads and an order-of-magnitude slowdown in performance (see Section 2.1 for a discussion of why this happens). This effect becomes more pronounced if the system needs to aggregate or batch requests to mitigate RDMA's request-level bottlenecks [47].

In a sense, software-level memory disaggregation, a technique initially intended to allow better specialization of servers for either compute or memory, instead imposes substantial compute-side CPU utilization whenever memory is needed. These overheads negate a significant fraction of the possible benefits of software-level disaggregation and, as others have also noted, can reduce the cost-efficiency of such systems compared to traditional architectures [26, 48, 50].

In this paper, we propose Cowbird, a method for offloading the administration of memory disaggregation from the compute node, thus enabling their CPUs to focus on their intended task: computation. In Cowbird, the offloaded tasks are, instead, the responsibility of devices like programmable switches [3] and SmartNICs [1, 15] (which are extremely power efficient on a per-byte basis [30]) or otherwise untilized CPU resources (which are ubiquitous in modern clouds [6, 47]).

At the center of Cowbird is an I/O abstraction that—without architectural support—allows applications to initiate remote memory requests using purely local operations, which are then serviced asynchronously by remote devices. Cowbird's API is agnostic to most properties of the offload platform and is, therefore, compatible with a broad range of widely available commodity hardware. We demonstrate this using two different implementations of the Cowbird abstraction: one using programmable switches and the other using spot-instance VMs.

Cowbird's design leverages the ability of modern remote and in-network compute to generate, read, and modify packets at high rates. Using this ability, an offload platform running Cowbird can continually poll compute-node memory and generate the necessary RDMA requests/responses to service any outstanding requests; in the meantime, the compute node can use the freed CPU cycles to process other tasks in a pipelined fashion. Of course, a naïve implementation of this mechanism on devices with limited computational capability and little-to-no visibility into compute-side behavior can lead to significant inefficiencies and performance bottlenecks. Cowbird, thus, introduces several novel techniques for batching, prioritization, and failure handling for efficient and robust data transfers.

Note that while Cowbird's generated messages impose network overheads, prior work has shown that networks spend much of their time idle [7, 37, 45] and devices can utilize those idle periods with little-to-no impact on user traffic or power draw [44]. Similarly, although Cowbird adds computation to outside components, offload platforms are typically much cheaper per byte processed [30], harder to monetize [6], and most importantly, out of the way of users' computations.

Our evaluation further shows that, with careful design, Cowbird can service typical application remote memory transfers at a rate that is indistinguishable from a purely local version. Cowbird eliminates CPU overheads, which unlocks application throughput improvements of up to 9× versus one-sided RDMA [27] and 1.6× versus Redy [47], a recent software-level disaggregation system. More specifically, this paper makes the following contributions:

- We propose Cowbird, a memory disaggregation architecture that converts remote memory accesses into local memory accesses without CPU architecture support by offloading the data transfers.
- We show that Cowbird is general to several hardware settings by implementing it on both Tofino programmable switches and remote servers.
- Finally, we adapt FASTER [27], a production open-source KV store, to use Cowbird. We show that with Cowbird, it can achieve the same throughput as in-memory execution when the working set is larger than local memory.

2 BACKGROUND AND MOTIVATION

We begin by providing background on both existing approaches for memory disaggregation and the opportunities provided by spot and in-network computing.

2.1 Memory Disaggregation

Memory disaggregation is a design methodology in which applications can leverage physically distinct memory, e.g., deployed to dedicated memory pools or harvested from remote servers. The potential benefits of this approach are broad and well-covered by prior work [4, 8, 11, 17, 20, 24, 32, 35, 40, 47]. As previously mentioned, proposals for memory disaggregation generally fall into two categories: hardware-level memory disaggregation and software-level disaggregation.

Hardware-level disaggregation is exemplified by the nascent CXL standard [2], which defines a communication mechanism between CPUs and physically distinct memory. CXL uses PCIe 5.0+'s physical-layer interfaces but replaces the traditional PCIe protocols and backplanes with CXL protocols and switches that provide low-latency and cache-coherent access to a slightly expanded pool of CXL-accessible memory. A goal of this approach is to hide the complexity of disaggregation behind traditional LD/ST instructions.

Software-level disaggregation, in contrast, extends the benefits of memory disaggregation to existing systems/hardware [17], further afield stranded memory [47], and application-specific requirements [32, 46]. Unfortunately, while software-level disaggregation broadens the applicability of disaggregation, prior work has shown

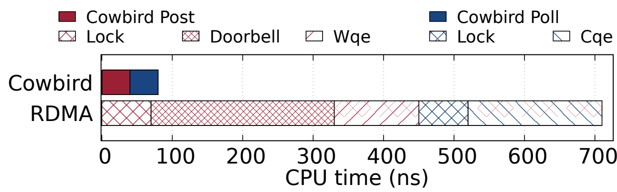


Figure 2: A breakdown of the compute-side CPU time of a single Cowbird read versus that of an asynchronous one-sided RDMA read. Red indicates a post task; blue a poll task. Note that network delay is not included in this metric—when the poll is called, the result is immediately available.

that it often incurs substantial overheads that partially offset any increased elasticity and reduced resource fragmentation [48].

As shown in Figure 1, a key component of that overhead is the CPU cost of initiating requests to remote memory. This is true of kernel-based software-level disaggregation mechanisms like LegoOS [35], which executes a remote memory access as part of every page fault, and Infiniswap [17], which treats remote memory as swap space and also uses a separate daemon process to manage remotely accessible memory. Both approaches require CPUs to block during remote calls to ensure page table consistency. Even for systems [32, 47] that do not block, however, the overhead of RDMA calls can still be high. Figure 2 shows the total compute-side CPU time of a read in both Cowbird and asynchronous one-sided RDMA in the testbed described in Section 7. Note that, in this experiment, the `ibv_poll_cq()` is called after the read completes, i.e., the latency is for a single check of the completion queue and shows the minimum CPU overhead of communication.

In total, RDMA requires an order of magnitude more CPU time than local memory writes and Cowbird. Although RDMA operations are conceptually simple (the post writes to an RDMA request queue pair and rings a doorbell register; the `poll` reads from a completion queue), these operations take significant time, as indicated by the detailed breakdown of Figure 2, obtained via `rdtsc` instrumentation of the Mellanox OFED driver. The reason is that each of the above subtasks requires spinlocks, atomics, and/or multiple expensive `mfence`/`sfence` instructions to ensure proper ordering of queue and doorbell register accesses.

In the end, while existing systems significantly outperform prior work and provide a number of important features, we argue that they have all been hamstrung by the simple need to invoke RDMA functions, often on a per-access basis.

2.2 Remote and In-network Computation

Parallel trends have led to the emergence of computational resources separate from the traditional server architecture. These compute resources are typically more constrained than traditional CPUs (e.g., in their capabilities, performance, or availability) but are also more cost-effective in terms of throughput per dollar [30].

In-network computation. One class of compute resource includes devices like programmable switches or SmartNICs. For example, P4-programmable Reconfigurable Match Table (RMT) switches have recently sparked development of a growing spectrum of in-network functions from fine-grained network monitoring [38, 42, 43] to application-specific acceleration [30, 34]. RMT switches provide the abstraction of a pipeline of 'reconfigurable' match-action table stages. Packets are processed sequentially such that each stage only works on one packet at a time, although different stages are pipelined. While not a Turing complete execution model, for applications that can fit their logic into an RMT pipeline, programmable switches provide a cost-effective [30], line-rate platform for executing computation inside the network.

VM type	On-demand price	Spot price
GCP: c3-standard-4	$0.257/h	$0.059/h
AWS: m5.xlarge	$0.192/h	$0.049/h
Azure: D4s-v3	$0.236/h	$0.023/h

Table 1: On-demand price and Spot instance price for VMs with 4 vCPUs and 16 GB memory on different cloud platforms. Data is from July 24, 2023.

A similar platform for in-network computation is SmartNICs. SmartNICs differ from traditional NICs by incorporating additional programmable processing components (e.g., FPGAs and/or multi-core SoCs) such that packets can be directed to the programmable elements for more advanced packet processing logic. Like with programmable switches, operators have found that offloading amenable tasks to these devices results in comparative cost savings and added compute capacity for paying users [12, 15, 22, 25].

Harvested CPUs. Finally, in addition to the newfound programmability of the devices described above, we note that many cloud providers have found ways to expose existing, unused compute capacity [6, 46, 47] to users. In fact, recent work has shown that even during periods of higher compute utilization, around 18% of CPUs in Microsoft's Azure go unallocated and unused [46]. Spot VMs are one way in which cloud providers expose these resources in an attempt to reduce waste. Spot instances provide a similar abstraction and capabilities as standard instances but at a significantly reduced cost. In return, the instances can be reclaimed by the cloud provider at any time, although generally with a grace period of several minutes and (anecdotally) a high likelihood of obtaining a new spot allocation if one is requested. In Table 1, we collect the price information for general-purpose VMs on different cloud platforms. With spot instances, the cost can be reduced by up to 90%, which makes even small improvements to compute-node CPU utilization worth it, especially if these instances can handle multiple compute nodes simultaneously. Some cloud platforms like GCP further provide pure spot CPUs with even lower prices: $0.009638 per vCPU-hour.

3 DESIGN OVERVIEW

The core idea behind Cowbird is to offload the transfer of memory away from the compute node. From the compute node's perspective, issuing a request in Cowbird involves only writes to local memory, and asynchronous retrieval of its results requires only a corresponding read. The overhead of RDMA is eliminated from the compute node, thereby freeing the node's CPU cycles to work on applications' compute tasks.

Instead, the data transfer is performed entirely by an offload engine, which (a) ensures timely processing of user requests, (b) implements batching and data packing to minimize load on the

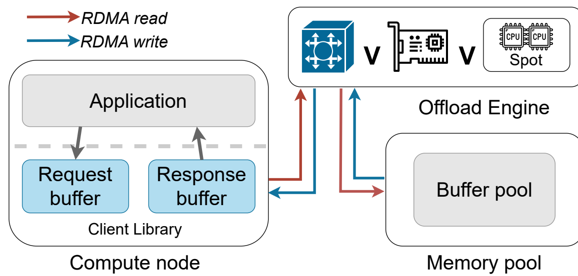


Figure 3: The architecture of a Cowbird system. An offload engine is responsible for actually executing data transfers between the compute node and memory pool.

compute-side RNIC (in addition to reducing load on the compute-side CPU), and (c) guarantees that all of the above respects sequential consistency even in the presence of multi-threaded applications.

The primary challenge in Cowbird's design is the definition of an interface and set of offloading protocols that are amenable to offload on a wide range of offload engines, some of which were not designed for uses like Cowbird but rather for more traditional packet processing.

System components. Figure 3 illustrates the high-level architecture of Cowbird. A Cowbird deployment consists of three main components:

- The *compute node* is the machine that executes the user's application alongside the Cowbird client library. The library interacts with request and response queues in local memory using simple load and store instructions.
- The *memory pool* is the device that hosts the pool of remote memory. This memory can be reserved or harvested from fragmented resources [47] but should be registered with the compute node client library.
- Finally, an *offload engine* executes the compute node's requested transfers without compute-node intervention. The engine generates/modifies RDMA messages that poll the compute node queues for new operations.

Figure 3 also sketches the flow of a typical Cowbird request between these components, which includes: (1) the client library writing to a lock-free request buffer, (2) the offload engine discovering the request and executing the requested transfer using spoofed, asynchronous RDMA messages, and (3) the offload engine posting the results to a response buffer.

While the Cowbird offload engine can be implemented on any platform that can generate and modify RDMA packets, in this paper, we focus on the possible variants on two contrasting platforms: P4 programmable switches (Section 5) and harvested spot VMs (Section 6). Our goal is not to advocate for one over the other—each variant has a different set of tradeoffs, and we anticipate that the ideal variant will depend on the specifics of each network operator. Instead, our goal is to demonstrate that it is possible to excise disaggregation from the compute-node CPU and that the Cowbird architecture is a general and effective way of doing so.

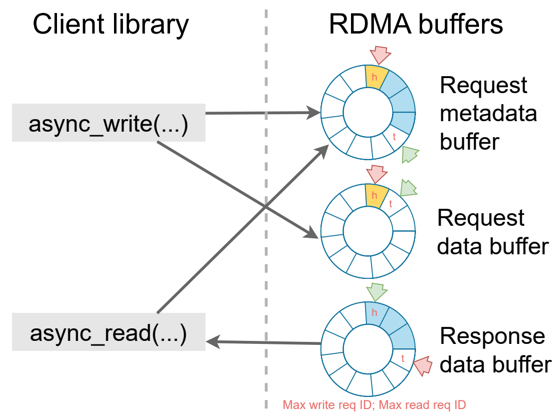


Figure 4: Relationship between client operations and the three compute-side buffers. *h* and *t* indicate the head and tail pointers. Red pointers and counters are packed into a contiguous memory block and updated by the offload engine. Green pointers are also packed and read by the offload engine with a single request.

4 THE COWBIRD COMPUTE NODE

We begin by describing the Cowbird API and client library on the compute node, whose operation is independent of the Cowbird offload platform and its specific capabilities.

4.1 The Cowbird API

Applications use Cowbird through the API calls listed in Table 2. Chief among them are two calls, `async_read()` and `async_write()`, which initiate asynchronous reads and writes to remote memory, respectively. All references to remote memory are expressed as an offset from the base `memory_pool_addr` of the allocated remote memory region, which is configured when the region is initialized.

Both the read and write functions return a request ID that can be used to retrieve results later. To that end, Cowbird provides an epoll-like interface for using those request IDs to check for completions efficiently. Users first `poll_create` a notification group for their requests, `poll_add` request IDs to that group, and then `poll_wait` for completions. Simple extensions can be made to the API to allow convenience methods like traditional `select/poll` semantics or an implicit notification group tied to each read and write. Note that, unlike actual `epoll` calls, Cowbird imposes some ordering constraints on the execution order of requests and returned responses, e.g., between the same operation types when issued by a single thread or when there are read-after-write dependencies to ensure linearizability.

4.2 Data Organization

Under the hood, `async_read()` and `async_write()` calls append request objects to local-memory queues, while `poll_wait()` checks the response buffers, also in local memory. A naïve implementation of these queues might simply marshall each request and its payload into a contiguous queue item, and do the reverse on the response-side. Unfortunately, each request can be of variable size, which interacts poorly with offload platforms that are optimized for

API	Description
`async_read(region_id, src, dest, length)`	Asynchronous call to read data from a remote source address to local destination with specified length. Returns a request ID.
`async_write(region_id, src, dest, length)`	Asynchronous call to write data from a local source address to a remote destination with specified length. Returns a request ID.
`poll_create()`	Initialize a notification group for Cowbird requests. Returns a poll ID.
`poll_{add/remove}(poll_id, req_id)`	Add/remove a request to/from an existing `poll` notification group.
`poll_wait(poll_id, responses, max_ret, timeout)`	Wait until either we receive `max_ret` completions or hit the timeout.

Table 2: Cowbird's simple, user-space API, shown here in C++.

packet processing, e.g., due to complex conditionals in request parsing and the possibility of segmentation. More generally, Cowbird requires the following of its data organization scheme:

R1 *Efficient processing by packet-centric devices:* Requests must be easily parsed, batched, and parallelized without the need for complex conditionals (e.g., to handle arbitrarily spaced header content).

R2 *Lock-free coordination:* Cowbird must avoid the expensive coordination required by the more general RDMA API, while guaranteeing consistency between application threads and the offload platform.

R3 *Minimized RDMA message count requirements:* Finally, to mitigate the latency of moving RDMA logic off of the compute node, Cowbird must minimize the number of RDMA operations per application-level request.

Request queues. On the request side, Cowbird achieves R1/2 using two separate physical data structures: one for fixed-size request metadata and the other for the associated data in the case of a write request. Both are laid out as per-hardware-thread, lock-free circular buffers whose structures are depicted in Figure 4.

The anatomy of a single entry of the request metadata buffer is shown in Table 3. `rw_type` is a value to indicate whether the request is valid and, if so, a read or write (padded to ensure alignment). `req_addr` represents the address to retrieve the data. For a read request, `req_addr` will be a valid address from the memory node. For a write request, `req_addr` will be a valid address from the compute node. `resp_addr` represents the address to write into after the data is retrieved from `req_addr`. The fourth field is `length`, which is the length of the real data to be read/written. Finally, the `region_id` uniquely identifies the target memory block.

The data buffer that stores payloads for write requests, in contrast, contains entries of variable length without any per-request metadata. Rather, clients append to-be-written data to the circular buffer in its raw form and will reference the region in the request metadata.

Response buffers. The response data buffer follows the format of the request data buffer. Raw data from reads are appended directly to the buffer without any per-response metadata; writes are not reflected in the circular buffer at all. Instead, the progress of operations is tracked by two fields:

- *Write progress:* Request-id of the last completed write.
- *Read progress:* Request-id of the last completed read.

Because Cowbird guarantees per-type linearizability, these are sufficient to track the progress of the offload engine.

Field	Bits	Valid domain
`rw_type`	16	compute and memory
`req_addr`	64	memory(read); compute(write)
`resp_addr`	64	compute(read); memory(write)
`length`	32	compute and memory
`region_id`	16	compute and memory

Table 3: Fields in a Cowbird request metadata block.

Bookkeeping. To reduce the message count (R3), all bookkeeping data (i.e., head/tail pointers and progress tracking) are packed into a contiguous memory region indexed by the writer/reader. The colors of metadata in Figure 4 illustrate this categorization, which ensures that all relevant metadata can be read/written with a single RDMA request. Offload-engine batching (described in Section 6) will further reduce the number of underlying RDMA messages under load.

4.3 Client Library Operation

The client library is responsible for the compute-side functions of the Cowbird API, which involves copying the requests from application threads into the request buffers and copying the responses back from response buffers while maintaining bookkeeping for both. The library code only executes when the application invokes a Cowbird API; there are no background operations.

Issuing a request. Upon invocation of an `async_read` or `write`, Cowbird executes a set of local-memory writes to prepare the request metadata and append the information to the appropriate queues. Briefly, for a read request, the library (1) atomically increments the request metadata tail pointer, (2) atomically increments the response data tail pointer, and (3) populates the five fields of Table 3 in the newly reserved request metadata entry appropriately. The `rw_type` cache line is written last and signals that the request is ready to execute. With x86-TSO, this sequence of atomic increments and writes guarantees consistent request issuance even without explicit locks or `mfence` instructions. Notably, the ordering of entries in the metadata and data buffers can differ, but their content remains consistent. Writes are similar but reserve and fill in an entry in the request data buffer instead of the response data buffer.

If, at any point, there is insufficient space in any of the queues or buffers, the library will return an error indicating that the application should retry later. In the case of a write, the retry can be immediate; in the case of a read, the application should process existing reads to clear buffer space before continuing. New requests cannot be queued until these operations are fully issued.

Headers	Packet type	Fields
BTH	All RDMA packets	opcode, QPN, PSN
RETH	RDMA read request RDMA write request	virtual address, remote key, length
AETH	RDMA read response RDMA acknowledgment	syndrome, MSN

Table 4: RDMA headers used by Cowbird-P4.

Handling responses. Clients of Cowbird are presented with an asynchronous communication abstraction. After filling the network delays with computation or pipelined request calls, client applications process responses by invoking a Cowbird API call, eliminating the need for interrupts or context switches. As mentioned, Cowbird processes requests of the same type and from the same region in linearized order, so the progress counters introduced in Section 4.2 can fully determine the set of completed responses. Moreover, tracking and polling for completions become very efficient operations in this model.

`poll_create()` allocates a list of `(region_id, req_id)` tuples. Adding or removing requests from the notification group updates an integer for the associated region that tracks the *maximum* registered `req_id`. The system knows that requests are complete when the response buffers' write and read progress indicators surpass each request, and it checks for such completions in every `poll*` call. For efficiency, `req_ids` are generated to encode their operation type, region id, and the incremented per-request id such that almost all checks can be done with simple integer arithmetic and comparison.

5 COWBIRD-P4 OFFLOAD ENGINE

In this section, we provide a proof-of-concept offload engine implementation, Cowbird-P4, that leverages programmable network devices adjacent to the compute node. Cowbird-P4 provides an interesting case study as these devices provide some of the most restrictive execution models among available offload platforms. At the same time, using these devices allows Cowbird-P4 to fully offload the movement of data from comparatively more expensive general-purpose CPUs.

Regardless of platform, we define several critical requirements of any Cowbird offload engine implementation:

- S1 *High maximum request rate:* For both throughput and latency, the offload engine should be able to poll the client-side request queues at a high rate.
- S2 *Minimized per-request message overhead:* Related to S1 and R3 of Section 4.2, the engine should also minimize the message overhead of each application-level request. In Cowbird-P4, this includes ensuring no recirculation overhead and promoting batched fetches of requests.
- S3 *Consistency:* In cooperation with R2, a Cowbird offload engine should provide consistency guarantees. As mentioned, Cowbird provides a particularly strong level of consistency—linearizability.
- S4 *Fault tolerance:* Finally, the engine should be resilient to congestion and occasional packet drops.

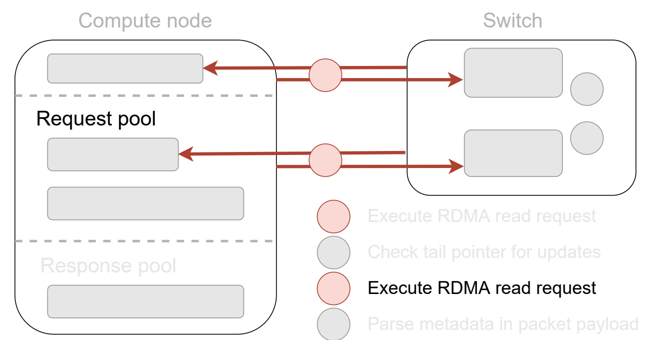


Figure 5: Cowbird-P4 probe phase procedure.

5.1 Wire Format

Cowbird-P4 is built on top of the RDMA over Converged Ethernet version 2 (RoCEv2) protocol [10]. RoCEv2 allows RDMA packets to be carried across Ethernet networks and processed by Ethernet switches. Switches are able to generate and modify these packets at line rate, which consist of the following headers, in order: Ethernet, IP, UDP, and base transport (BTH)[1]. They may also contain RETH and AETH headers, depending on the type of the packet (see Table 4 for the relevant headers and contents).

5.2 Communication Protocol

The Cowbird-P4 protocol consists of four phases: Setup, Probe, Execute, and Complete. The first phase occurs when the application starts. The last three occur on a per-request basis. For simplicity, we first consider a single request in isolation and generalize in Sections 5.3 and 5.4.

Phase I: Setup. The compute node and memory pool will begin by initializing an RDMA connection and registering buffer memory regions on both sides of the connection. The compute node will then send the switch configuration information through an RPC endpoint running on the switch control plane, i.e., the QP numbers; the current PSN for each QP; and the base memory addresses, remote keys, and total size of all registered memory regions. The switch will use this information to allocate the required register space and reconfigure packet generation to supply RDMA packets of the format specified in Table 4. Modifications or termination of the channel also occur through this interface.

Phase II: Probe. In the Probe phase, the switch periodically generates RDMA read requests to check whether the tail of the compute node's request metadata queue has moved and new requests become ready to process.

Modern switches can generate packets quickly enough to saturate all outgoing links with probe packets; however, doing so could result in high bandwidth overheads. To mitigate potential overheads, Cowbird-P4 configures the probes with the lowest priority across the switch pipeline (i.e., in the ingress arbiter, traffic manager, and egress arbiter). Prior work [44] has shown that with proper priority settings, low-priority packet injection has little-to-no impact on user network traffic or switch power consumption. It further

[1] Current programmable switch implementations cannot compute RDMA iCRCs, so like [34], Cowbird-P4 disables these checks on the end host; however, this limitation is not fundamental.

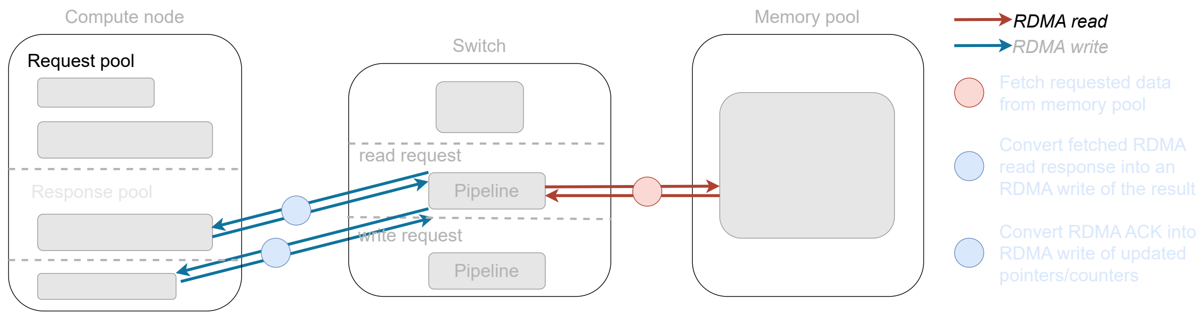


Figure 6: Operation of the Cowbird-P4 Execute and Complete phases for read requests.

limits probe rates to a configurable application-specific expected host-level I/O throughput (1 probe per 2 µs for our prototype implementation of FASTER). The probe sizes are small enough that the worst-case memory bus overhead is less than 0.1%. Note that the switch can also start at a low baseline rate and ramp up only when activity is detected, allowing users to tradeoff extra probe memory accesses with worst-case completion latency while maintaining high throughput.

The switch tracks its view of the head and tail pointers in stateful data-plane registers. For each probe response, the switch will compare the received tail pointer with the previous value and, if different, it will issue one or more RDMA read requests for the contents of the request metadata queue (head→tail). Specifically, the switch will take the probe response, recycle it by removing the AETH header and adding a RETH header (creating an RDMA read request), and then use it to read starting from the local head pointer.

Figure 5 visualizes the procedure for retrieving new request metadata. The Cowbird-P4 switch checks for new requests in Steps 1 and 2. If the request metadata tail pointer has advanced, it issues an RDMA read in Steps 3 and 4.

Phase III: Execute. After the switch receives the metadata for a new request, it enters the Execute phase to execute the transfer. Depending on the `rw_type`, the protocol diverges.

For read requests, the protocol follows the first two steps in Figure 6. In Step 1a, the switch recycles the RDMA read response packet from Phase II to—without relying on packet generation—create a new RDMA read request that will fetch the requested data from the memory pool; it can craft the content of this packet using only the request metadata and stored connection state from stateful data plane registers. In addition to sending this packet to the memory pool, Cowbird-P4 also stores the target response address in a hash table so that it knows where to write the data in the subsequent step.

The memory pool responds to the Step 1a request like any other RDMA read. Although the switch cannot parse the contents of the response (due to PHV limitations), Cowbird-P4 can, similar to Step 2a, recycle the read response to create an RDMA write request with a new header and the unmodified read-response payload as the contents to write. Note that when the requested data size is larger than 1024 bytes, RDMA will automatically segment the response into RDMA Read Response First, Middle, and Last packets. Cowbird-P4 will convert them into the corresponding RDMA Write packets: Write First, Middle, and Last.

Write requests in Cowbird are executed similarly, but with the source and destinations of all messages reversed. The process is illustrated in Figure 7 (Steps 1b & 2b).

Phase IV: Complete. After Cowbird-P4 finishes transferring the data in Phase III, the final step is to update counters and pointers in the compute node to signal completion and enable overwriting of old requests.

Cowbird-P4 does this by sending an RDMA write request to the compute node (again, recycling the previous RDMA response/acknowledgment) to update the request metadata/data head pointers, the response data tail pointer, and the read/write completion counters using current values. All the pointer and counter updates can be applied using a single RDMA write request to the contiguous memory block mentioned in Section 4.2. This process is identical for Cowbird reads and writes (Step 3 in both Figures 6 and 7).

5.3 Consistency and Fault Tolerance

The above procedure assumes a single `async_read/write` call and no packet drops. In practice, packets from all three phases may be in flight concurrently, and some can be dropped. For example, if the switch queries the request metadata queue and receives a new request, i, as part of the Probe phase, it will convert the packet to a Phase III Step 1 RDMA read request. While that request is still in flight, the switch will issue another Probe request packet to check for request $i + 1$, which may also advance to further phases before i completes.

Consistency. In the face of concurrency, Cowbird-P4 ensures linearizability. To see how, we first note that the switch probes all requests in FIFO order and interdicts all RDMA operations. Thus, the programmable switch's data plane pipeline serves as a serialization point for all requests.

Within a request type (read or write), execution will never be reordered from the data plane initiation order. Across request types, however, some reordering is possible, e.g., when the switch-to-memory path is congested. In this case, a naïve implementation may cause Phase III Step 1b to become delayed, causing subsequent reads to fetch stale data. In general, Cowbird needs to halt processing of newly probed reads only during the execution of writes on an overlapping address in Phase III Step 1b. Cowbird-Spot takes exactly this approach. Unfortunately, current programmable switches struggle to implement the range queries necessary for that logic. Instead, Cowbird-P4 temporarily pauses the processing of all newly probed reads to maintain linearizability. Other instances can keep

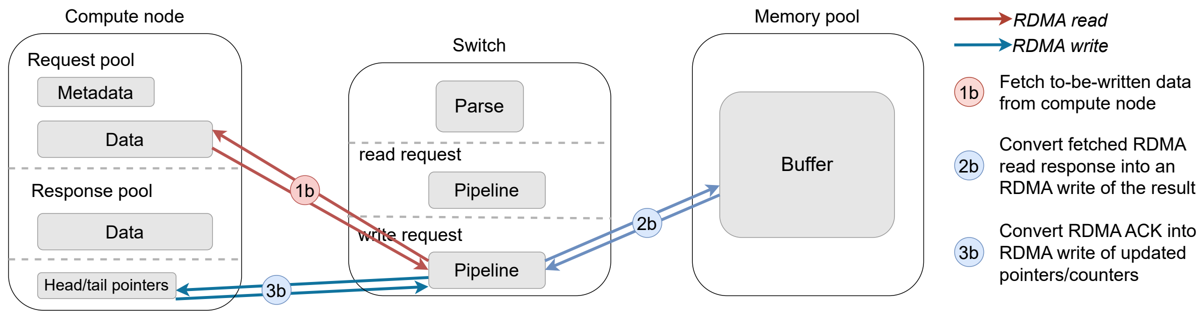


Figure 7: Operation of the Cowbird-P4 Execute and Complete phases for write requests.

the switch busy during these periods (see Section 5.4). Step 2b and subsequent operations are not explicitly synchronized as they will be serialized by the switch/RNIC.

Fault tolerance. With PFC enabled, packets in Cowbird-P4 can be lost due to corruption. Without PFC enabled, congestive losses may also be possible, primarily in Phase III between the switch and memory pool as the switch and compute node are adjacent. However, regardless of the location and cause of the loss, Cowbird can recover using data plane timeouts and retransmissions triggered by the periodic locally generated packets. From the switch's perspective, losing an outgoing packet toward either machine will result in PSN desynchronization and all subsequent packets being rejected. In contrast, the loss of an incoming packet will appear as a gap in the execution history. In both cases, Cowbird-P4 can detect a timeout and utilize a Go-Back-N approach by resetting the local head pointer and PSN and re-executing the Probe phase starting from that point.

5.4 Handling Multiple Cowbird Instances

While different threads and memory regions can use a single set of compute-node request/response queues, there may be instances where multiple sets of queues are required (e.g., for isolation or multiple compute/memory node pairs).

The Cowbird-P4 switch handles requests from different instances using time-division multiplexing. Specifically, in the Probe phase, the switch will cycle between all registered instances in a round-robin fashion. Note that more complex policies are possible, e.g., to prioritize more active applications; however, we leave a full exploration of those policies to future work. Note also that multiplexing may not be necessary if the instances are on servers are connected to different ports—the Cowbird-P4 switch can generate probes at the maximum rate for every compute-facing physical port.

Non-Probe phases of the protocol proceed normally, triggered as before by incoming RDMA requests/responses. The main challenge is that, while the request metadata fetched in Phase II will include the instance ID in the request contents, subsequent packets will not. Packet tagging is not possible as Cowbird-P4 must be compatible with existing RNIC implementations. Instead, Cowbird-P4 stores a QPN-to-instance-ID mapping, which it queries at every step.

6 COWBIRD-SPOT OFFLOAD ENGINE

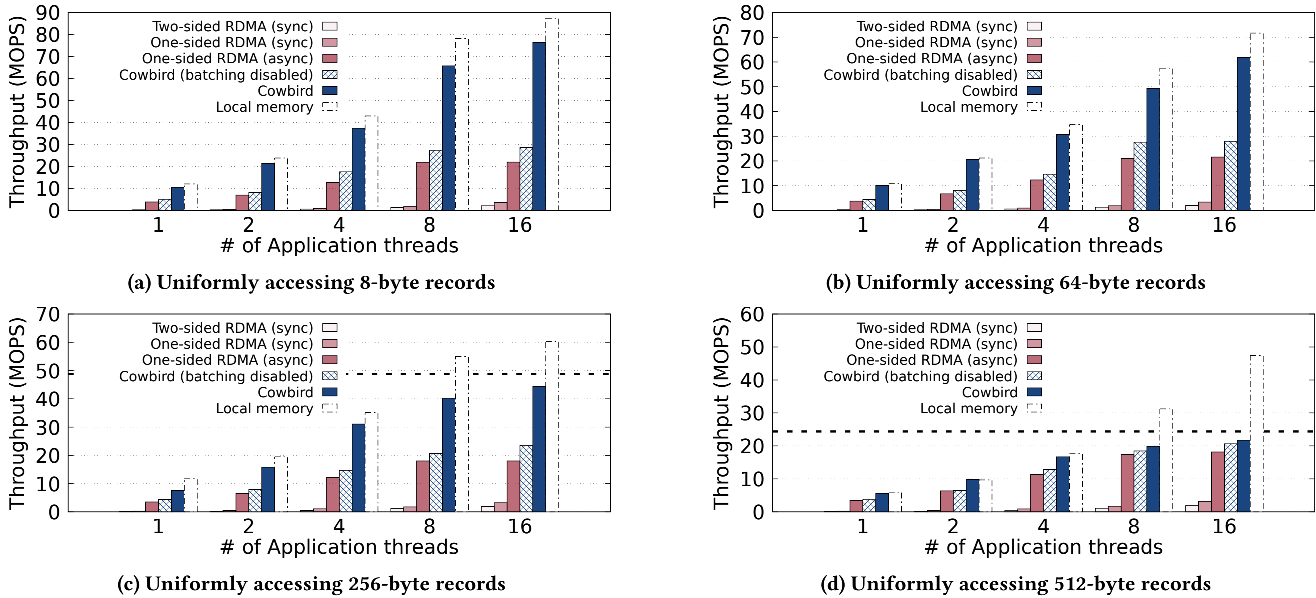
We present a second variant of the Cowbird offload engine, Cowbird-Spot. Compared to Cowbird-P4, Cowbird-Spot sits on the other end of the spectrum and utilizes general-purpose processors to offload memory transfers. These compute resources can come from many different sources, e.g., the ARM cores of a SmartNIC [1], the management CPU of a harvested-memory VM [47], or a separate spot instance dedicated to data-transfer offload. For simplicity, we assume the environment of [47], but the design should be generalizable. The distinguishing feature of this class of implementations is its ability to perform arbitrary computations and manipulate temporary local memory. Based on our evaluation, Cowbird-Spot offload requires minimal CPU capacity.

Request processing. The high-level protocol of Cowbird-Spot is nearly identical to that of Cowbird-P4—it implements all the steps of Section 5.2, but instead of generating and recycling raw packets in the switch data plane, it initiates RDMA operations through traditional, host-level RDMA interfaces with an event-driven agent process running on the offload processor. At a protocol level, the primary differences between the two variants occurs in Phase III. The agent parses the fetched metadata requests and executes the requested transfers through a series of RDMA requests. Unlike Cowbird-P4, the processor can perform a simple check for overlapping memory ranges so that it only needs to pause per-thread reads when absolutely necessary for consistency. Also unlike Cowbird-P4, the offload processor can batch `BATCH_SIZE` read responses in its local memory before issuing a single RDMA write for the whole batch to the compute node in Step 2a. Batching in this manner further reduces the load on the compute node and its network interface card. It also results in lower compute overhead on the offload engine as a result of issuing fewer RDMA calls. Both can improve the cost-efficiency of Cowbird.

7 CASE STUDY: THE FASTER KV STORE

We implement Cowbird on a testbed with a Wedge100BF-32X switch, which contains a Tofino programmable switch ASIC. The servers are equipped with Intel Xeon Silver 4110 CPUs (8 physical cores with hyper-threading) and 96 GB RAM and connected via 100 Gbps NVIDIA Mellanox ConnectX-5 NICs. In total, the Cowbird-P4 offload engine consists of ~1700 lines of P4 for the data plane and ~500 lines of Python for the control plane. Cowbird-Spot contains 2000 lines of C++.

As a proof of concept, we port FASTER, an open-source key-value store system from Microsoft, to use Cowbird. FASTER is an attractive platform as: (1) it represents a production system that can directly benefit from increased memory capacity, and (2) it already supports memory disaggregation with Redy [47] that is based on compute-initiated RDMA.



(a) Uniformly accessing 8-byte records

(b) Uniformly accessing 64-byte records

(c) Uniformly accessing 256-byte records

(d) Uniformly accessing 512-byte records

Figure 8: Hash table performance backed by disaggregated memory. Dashed lines in (c) and (d) represent the upper-bound by bandwidth. Cowbird closes the gap between remote and local memory until the bandwidth limit.

Records in FASTER are stored in a *hybrid log*—a log partitioned across main memory (the tail of the log that is writable) and storage (the read-only part of the log). A read operation to the key-value store first looks up the log address in a hash index and then retrieves the record from either main memory or storage. For the insertion operation, the record is first appended to the tail of the hybrid log and added to the index. When main memory is insufficient, older data will be appended to storage, e.g., SSDs or remote memory.

We adapt FASTER to use Cowbird by instantiating an `IDevice`, the interface FASTER exposes for implementing its storage layer for the larger-than-memory part of the log. To reduce contention, each FASTER thread calls through the device `poll_create()` to create a notification group. After issuing an I/O operation with `async_read()` or `async_write()`, a thread immediately calls `poll_add()` to add the request to the notification group and invokes `poll_wait()` periodically to complete pending requests. The simple interface of Cowbird makes the integration straightforward.

8 EVALUATION

We validate Cowbird by answering the following questions.

- Can Cowbird deliver its promise of focusing CPUs on the compute node on application workloads? What does it mean to application performance?
- How does Cowbird perform compared to state-of-the-art approaches to disaggregating memory?
- Does Cowbird impact latency in a negative way?
- What is its impact on network resources and bandwidth?

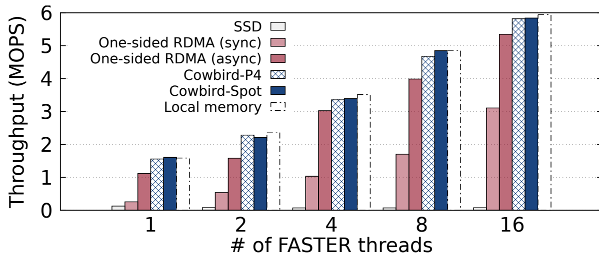
Methodology. Our evaluation consists of two workloads: (1) a throughput microbenchmark with a hash table where a hundred million records are split between compute-local memory (5%) and remote memory (95%) and (2) FASTER with the YCSB benchmark [13]. Recent production traces [9] show that record sizes in real-world key-value stores are generally small and mostly range from 8 to 512 bytes. In the microbenchmark, we compare Cowbird-Spot with different RDMA baselines (two-sided/one-sided verbs and synchronous/asynchronous I/O). To evaluate whether Cowbird efficiency generalizes to different offloading hardware, we include both Cowbird-P4 and Cowbird-Spot to speed up FASTER along with three baselines as follows.

- Secondary storage (the default storage backend in FASTER) that uses a local SATA SSD with 6 Gbs throughput on the compute node to store the read-only portion of the hybrid log.
- One-sided RDMA, an alternative design of an `IDevice` that can leverage remote memory using traditional one-sided RDMA verbs. This baseline does not assume any remote compute capabilities, so the compute node is responsible for all data transfers. We include both synchronous and asynchronous communication.
- Purely local memory that represents an upper bound on disaggregated memory performance, for which we create an `IDevice` that utilizes only compute-local memory and does not leverage any remote memory.

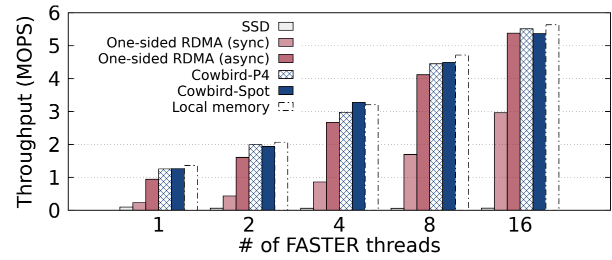
Unless otherwise specified, experiments were run on the testbed described in the previous section.

8.1 Remote Memory Performance

We first evaluate how the benefits of Cowbird's asynchronous I/O and communication offloading are reflected in overall application performance with disaggregated memory. Synchronous one- and two-sided RDMA issue one request at a time. Asynchronous one-sided RDMA issues requests in batches of size 100 and overlaps communication and computation, as does Cowbird. Results shown are for Cowbird-Spot but are identical for other variants.

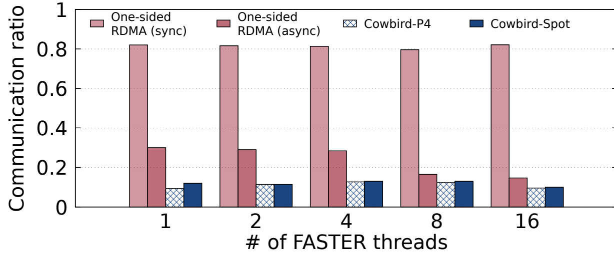


(a) 64-byte records

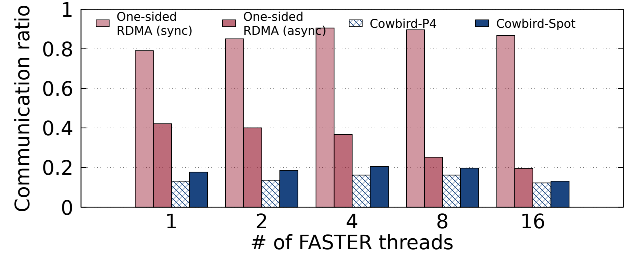


(b) 512-byte records

Figure 9: FASTER performance on YCSB (Zipfian θ=0.99) with Cowbird and baseline storage backends.



(a) 64-byte records



(b) 512-byte records

Figure 10: The effect of Cowbird's CPU savings for FASTER with Cowbird and baseline storage backends. The communication ratio is defined as the time spent in the communication library over the total execution time of the application.

Microbenchmark results. The hash table throughput test stresses Cowbird in two representative scenarios. When performance is limited by applications' own efficiencies (i.e., not network bandwidth), we expect remote memory with Cowbird to achieve similar performance as local memory. Otherwise, when applications spawn sufficient threads, Cowbird should easily drive the throughput to the network bandwidth bottleneck. Figures 8a and 8b confirm the first expectation, where the application accesses small (8 B and 64 B) records. These experiments show that (1) asynchronous I/O is an order of magnitude more efficient, (2) offloading communication with Cowbird brings additional performance win compared to asynchronous RDMA, and (3) batching with Cowbird is up to 3.5× faster than RDMA and closes the gap between local and remote memory performance (within 11.4%).

Figures 8c and 8d validate the other expectation—when accessing larger records, the application can fully utilize the network bandwidth with 16 threads using Cowbird. Note that for larger message sizes and thread counts, asynchronous one-sided RDMA can also eventually reach network bandwidth saturation but does so at much higher values of both. Thus, while applications with a consistent stream of large messages may not benefit significantly from Cowbird's CPU savings, compute-bound workloads will.

Benchmarking FASTER. We create YCSB databases with 8 B keys for both small (64 B) and large (512 B) values that contain 250 and 50 million records, respectively. The total data sizes in FASTER are 18 GB and 24 GB, and we configure FASTER to utilize 5 GB local memory for the tail of the log. The remaining data is stored in the `IDevice` instantiation and to serve skewed YCSB workloads. This configuration ensures that most operations are serviced by the storage layer (SSDs or remote memory) to stress the performance of Cowbird and the baselines. We scale application threads up to all available cores on the compute node. Figures 9 and 10 show the results for both database configurations.

One conclusion we can draw from Figures 9a and 9b is that utilizing remote memory for FASTER is at least 2.3× faster than SSDs. Cowbird further boosts this advantage: the speedup with Cowbird ranges from 12× to 84×. These results show that spilling state to remote memory results in significantly better performance than using secondary storage. In fact, compared to the performance of using local memory, we can see that Cowbird is consistently within 8% of that of local memory, validating the close-to-local-memory performance seen in the microbenchmark. A key reason for this performance benefit is the ability of Cowbird to reduce the time that the application spends on communication and reduce the performance overhead of disaggregation. Figure 10 depicts the communication ratio of the systems, defined as the time spent in the communication library over the total execution time of the application. FASTER with synchronous RDMA can spend more than 80% of its time on communication tasks, while Cowbird consistently spends less than 20%, with much of that in wrapper code. Note that in both Figures 9a and 9b, the relative overhead of asynchronous one-sided RDMA reduces with higher thread counts as the end-to-end performance bottleneck becomes FASTER's cross-thread coordination in `IDevice`. A more embarrassingly parallel application would exhibit performance closer to that of Figure 8.

Finally, we draw attention to the comparison between Cowbird-Spot and Cowbird-P4 to investigate the generalizability of Cowbird benefits to different offloading hardware. The figures show that these two approaches achieve similar performance across different workloads and scalability settings and improve FASTER throughput by up to 40% compared to asynchronous RDMA.

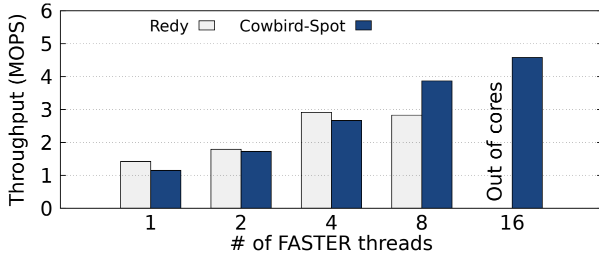


Figure 11: FASTER throughput with Cowbird and Redy.

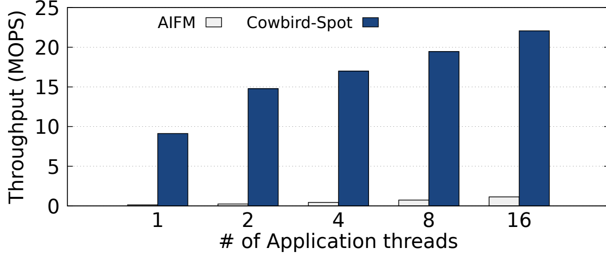


Figure 12: Throughput of uniformly reading 8 B objects from remote memory with Cowbird and AIFM.

In summary, our experimental results demonstrate that Cowbird successfully reduces the workload of the compute node CPUs, freeing them for processing application workloads.

8.2 Comparison with Other Approaches

In addition to comparing Cowbird to the alternative communication primitives in the previous subsection, we also compare Cowbird to two recent application-integrated disaggregation frameworks.

Cowbird versus Redy. Redy [47] is a system that utilizes remote memory as an in-memory cache by exploiting RDMA. To achieve high throughput, it batches user requests and sends them to the memory server through RDMA connections that are optimized for throughput. Upon receiving the batched requests, the memory server processes the requests sequentially and then writes back a batch of responses to the client. In optimizing performance, Redy spawns extra I/O threads that are pinned to physical cores on the compute node for batching requests and processing completions.

We run FASTER with Cowbird-Spot and Redy using the YCSB benchmark (64-byte records, uniform, 1 GB local memory). With Redy, the number of threads varies from 1 to 8 since Redy needs extra cores for its I/O threads. In fact, even when we allocate 8 cores to FASTER, the remaining cores are not sufficient for Redy to achieve its optimal performance, as shown in Figure 11. This experiment demonstrates Cowbird's benefits of saving more cores for applications.

Cowbird versus AIFM. AIFM [32] is another state-of-the-art userspace memory disaggregation solution. After sending a remote memory request, AIFM uses Shenango [29] to free the core and allow other threads to swap in. The original thread is scheduled again when the data is ready. For these experiments, we used an unmodified version of AIFM using their recommended deployment on CloudLab's xl170 instances. We deployed Cowbird-Spot on the same testbed with access to the same resources to ensure a fair

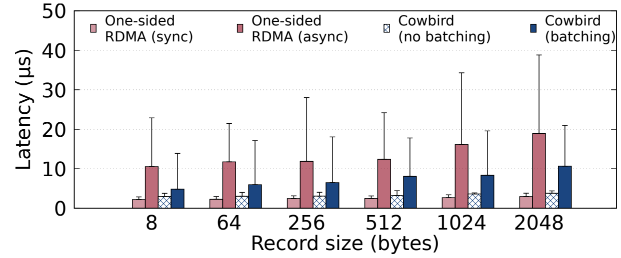


Figure 13: Cowbird-Spot and RDMA latency comparison. Bars and capped lines show the median and tail (p99).

comparison. We conduct experiments with random reads of 8-byte objects on both AIFM and Cowbird. Figure 12 shows that Cowbird achieves an order of magnitude (up to 71×) higher throughput across thread counts.

8.3 Cowbird Latency

As an asynchronous communication abstraction, Cowbird is primarily concerned with throughput, of both the remote memory accesses and application compute. To investigate Cowbird's potential impacts on latency, we compare its latency with one-sided RDMA for reading records of different sizes from remote memory. As seen in Figure 13, without batching, the latency of Cowbird's communication protocol is similar to that of traditional synchronous one-sided RDMA. Cowbird-Spot increases latency slightly due to 2 additional RTTs (to fetch and update head/tail pointers), offload engine processing delay, and the polling interval; however, it also offsets those increases by reducing the time it takes to post the request and poll the response. The net increase is minimal.

For Cowbird with batching and asynchronous one-sided RDMA, we continue to use the batching configuration of Section 8.1. As expected, these approaches increase both median and tail latencies, but Cowbird can achieve latency that is much lower than asynchronous one-sided RDMA ($< 10\,\mu$s and $< 20\,\mu$s for the median and tail, respectively).

8.4 Cowbird Overhead

Resource usage. Cowbird-P4 does not require any computation resources on either the compute or memory node. On the network device, our prototype implementation is optimized to fit into the switch resource constraints without packet recirculation. Table 5 details its data plane pipeline resource consumption, assuming the worst case where all ports are utilized for Cowbird-P4. The logic spans several stages, but the SRAM/TCAM usage within each stage is minimal, leaving space for concurrent Cowbird-P4 instances and other switch applications.

For Cowbird-Spot, we limit its resource usage to at most one CPU core. The prior evaluation results show that this configuration suffices for all the application threads.

Network bandwidth overhead. The bulk of Cowbird's network traffic is from Probes, but as these are configured with the lowest network priority, their impact on user traffic is minimal [44]. We note that Phase III packets also do not add overhead, at least when compared to other RDMA disaggregation solutions. The only extra traffic is from the periodic garbage collection messages of Phase IV.

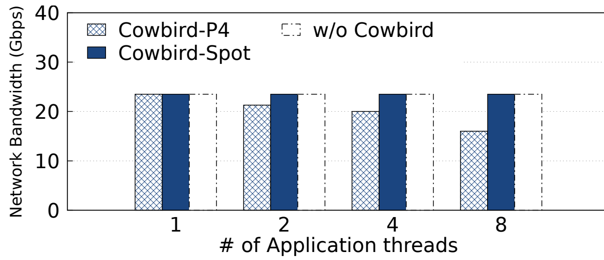


Figure 14: Bandwidth of contending TCP flows using 512 B records with Cowbird-P4, with Cowbird-Spot, and without Cowbird.

PHV	SRAM	TCAM	Stages	VLIW instrs.	sALU
1085 b	1424 KB	1.28 KB	12	38	11

Table 5: Cowbird data plane resource usage for a 32-port L3 forwarding Tofino switch.

To measure the total network overhead of Cowbird, we introduce contending applications of TCP transfers on the compute node where Cowbird runs concurrently. We configure an `iperf3` client with 10 threads on the compute node to continuously send traffic toward a third server (different from the memory node) with a 25 Gbps NIC and measure its bandwidth with and without Cowbird. To provide an upper bound on Cowbird's performance impact, we configure the RDMA packets with higher priority than the user traffic. Cowbird executes with 8 application threads.

Figure 14 shows the aggregate bandwidth of TCP flows when running Cowbird in FASTER with 512 B values. The overhead of Cowbird-Spot is negligible even with 8 application threads. With Cowbird-P4, TCP bandwidth drops by up to 30% in this worst-case scenario, which reflects the lack of response batching in the protocol.

9 RELATED WORK

Hardware/networking support. Memory disaggregation systems (include Cowbird) often rely on Remote Direct Memory Access (RDMA) [10, 18, 28] due to its high performance and the ubiquity of its support in modern NICs.

Compute Express Link [2] (CXL) has emerged as a potential alternative due to even lower latencies, lower CPU overheads, and integration with processor designs. Samsung introduced a prototype that uses CXL to expand local host physical memory [33]. More recently, DirectCXL [16] proposes to use the *CXL.mem* protocol to directly connect host CPUs to remote memory to execute load and store instructions with latencies lower than RDMA. However, at the time of writing, a far-memory-capable CXL fabric is neither fully specified nor commercially available. Cowbird can be seen as a way to achieve some of the benefits of CXL for far memory using only currently available hardware.

OS/runtime abstractions for disaggregation. A series of recent work abstracts away the complexities of underlying architectures by disaggregating the OS or managed runtimes, e.g., LegoOS [35], Infiniswap [17], and Semeru [40]. While useful for their backward-compatibility, these systems suffer from the issues in Figure 1 as well as other performance issues inherent in OS/runtime disaggregation, e.g., context switching overhead and read/write amplification [32].

Co-designed applications. Many systems, including Cowbird, instead bypass the OS and expose disaggregated memory directly in user space with new data structures, programming models, or remote memory libraries. Aguilera et al. [4] propose a set of data structures optimized for constructing efficient remote memory-aware applications. In addition to data structures, AIFM [32] also modifies the application runtime and introduces remote agents to further lower the cost of using remote memory. Redy [47] offers a simple, byte-addressable device abstraction for stranded memory or spot instances as high-performance caches, and LegoBase [50] and Sherman [41] optimize databases for disaggregated memory. Most of the optimizations in these systems minimize remote memory access latency or the volume of data movement, which is orthogonal to the goal of Cowbird—making sure applications use their CPUs to compute, not move data.

Compute offloading. Finally, several existing memory disaggregation systems offload subsets of their computation to improve performance. For example, Semeru offloads parts of Java GC, AIFM offloads light-weight data structure operations, and TELEPORT [49] pushes down memory-intensive operators. Similarly, StRoM [36] and Clio [19] offload to SmartNICs with FPGAs while MIND [23] chooses to offload the cache coherence protocolto programmable switches. RedN [31] modifies the RDMA driver and lifts the existing RDMA verbs to a set of programming abstractions that are Turing complete. Hyperloop [21] targets the replicated transactions of storage systems and offloads the CPU work from the critical path to the RDMA NICs.

Cowbird is distinct in its attempt to offload even the RDMA operations to remote compute, as well as its flexibility regarding the choice of offloading hardware.

10 CONCLUSION

We present Cowbird, a system that frees CPU cycles spent on accessing disaggregated memory. By doing so, it allows the compute node to allocate all CPUs for application workloads. The asynchronous I/O interface for requesting remote data and zero local CPU cost in executing the requests, enabled by compute offloading, make this work unique compared to other disaggregated-memory systems. The evaluation with a production key-value store shows that Cowbird-freed CPUs can significantly improve overall application performance.

This work does not raise any ethical issues.

ACKNOWLEDGMENTS

We gratefully acknowledge our shepherd Aditya Akella and the anonymous SIGCOMM reviewers for all of their help and thoughtful comments. We also thank Phil Bernstein for the valuable feedback on the early draft of this work. This work was funded in part by Samsung, Google, and NSF grants CNS-1845749 and CNS-2107147.

REFERENCES

[1] Bluefield smartnic. https://network.nvidia.com/files/doc-2020/pb-bluefield-smart-nic.pdf, 2022.

[2] Compute express link: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org, 2022.
[3] Intel tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, 2022.
[4] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*, pages 120–126. ACM, 2019.
[5] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 14:1–14:16. ACM, 2020.
[6] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing slos for resource-harvesting vms in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, November 2020.
[7] Alexey Andreyev. Introducing data center fabric, the next-generation facebook data center network. https://goo.gl/rE8wkL, 2014. Facebook.
[8] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. In Amar Phanishayee and Ryan Stutsman, editors, *12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2020, July 13-14, 2020*. USENIX Association, 2020.
[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.
[10] Broadcom. Rdma over converged ethernet (roce). https://techdocs.broadcom.com/us/en/storage-and-ethernet-connectivity/ethernet-nic-controllers/bcm957xxx/adapters/RDMA-over-Converged-Ethernet.html, 2022.
[11] Amanda Carbonari and Ivan Beschasnikh. Tolerating faults in disaggregated datacenters. In Sujata Banerjee, Brad Karp, and Michael Walfish, editors, *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*, pages 164–170. ACM, 2017.
[12] Adrian Caulfield, Paolo Costa, and Monia Ghobadi. Beyond smartnics: Towards a fully programmable cloud. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6. IEEE, 2018.
[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.
[14] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In Ratul Mahajan and Ion Stoica, editors, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 401–414. USENIX Association, 2014.
[15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.
[16] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.
[17] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 649–667. USENIX Association, 2017.
[18] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 202–215. ACM, 2016.
[19] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: a hardware-software co-designed disaggregated memory system. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 417–433. ACM, 2022.
[20] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In Dave Levine, Sachin Katti, and Dave Oran, editors, *Twelfth ACM Workshop on Hot Topics in Networks, HotNets-XII, College Park, MD, USA, November 21-22, 2013*, pages 10:1–10:7. ACM, 2013.
[21] Daehyeok Kim, Amir Saman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In Sergey Gorinsky and János Tapolcai, editors, *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 297–312. ACM, 2018.
[22] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 756–771, 2021.
[23] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: in-network memory management for disaggregated data centers. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 488–504. ACM, 2021.
[24] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In Stephen W. Keckler and Luiz André Barroso, editors, *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 267–278. ACM, 2009.
[25] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333. 2019.
[26] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 843–857. USENIX Association, 2020.
[27] Microsoft. Faster: Fast persistent recoverable log and key-value store + cache, in c# and c++. https://microsoft.github.io/FASTER/, 2022.
[28] Nvidia. Nvidia quantum infiniband platform. https://www.nvidia.com/en-us/networking/products/infiniband/, 2022.
[29] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 361–378. USENIX Association, 2019.
[30] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 194–206, 2021.
[31] Waleed Reda, Marco Canini, Dejan Kostic, and Simon Peter. RDMA is turing complete, we just did not know it yet! In Amar Phanishayee and Vyas Sekar, editors, *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 71–85. USENIX Association, 2022.
[32] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: high-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 315–332. USENIX Association, 2020.
[33] Samsung. Samsung unveils industry-first memory module incorporating new cxl interconnect standard. https://semiconductor.samsung.com/newsroom/news/samsung-unveils-industry-first-memory-module-incorporating-new-cxl-interconnect-standard/, 2021.
[34] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.
[35] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 69–87. USENIX Association, 2018.
[36] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: smart remote memory. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 29:1–29:16. ACM, 2020.
[37] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *SIGCOMM Comput. Commun. Rev.*, 45(4):183–197, aug 2015.

[38] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176, 2017.

[39] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 306–324. ACM, 2017.

[40] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 261–280. USENIX Association, 2020.

[41] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In Zachary Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1033–1048. ACM, 2022.

[42] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 402–416, 2018.

[43] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. 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*, pages 296–309, 2020.

[44] Liangcheng Yu, John Sonchack, and Vincent Liu. OrbWeaver: Using IDLE cycles in programmable networks for opportunistic coordination. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1195–1212, Renton, WA, April 2022. USENIX Association.

[45] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, pages 78–85, 2017.

[46] Qizhen Zhang, Phil Bernstein, Daniel S. Berger, Badrish Chandramouli, Boon Thao Loo, and Vincent Liu. Compucache: Remote computable caching using spot vms. In *Conference on Innovative Data Systems Research (CIDR 2022)*, January 2022.

[47] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. Redy: Remote dynamic memory cache. *Proc. VLDB Endow.*, 15(4):766–779, 2021.

[48] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Understanding the effect of data center resource disaggregation on production dbmss. *Proc. VLDB Endow.*, 13(9):1568–1581, 2020.

[49] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Optimizing data-intensive systems in disaggregated data centers with TELEPORT. In Zachary Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1345–1359. ACM, 2022.

[50] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proc. VLDB Endow.*, 14(10):1900–1912, 2021.