PD3: Prefetching Data with DPUs for Disaggregated Memory

Sidharth Sankhe
University of Toronto

Felix Zhang
University of Toronto

Umayrah Chonee
University of Toronto

Sherman Lim
National University of Singapore

Jiasheng Hu
University of Toronto

Jialin Li
National University of Singapore

Qizhen Zhang
University of Toronto

Abstract

We introduce PD3, a memory disaggregation solution that *avoids* cache misses, via prefetching, on compute servers and thus all their associated overhead. Unlike a traditional prefetcher that may pollute the cache or miss preloading opportunities due to false positives and false negatives, PD3 prevents mispredictions with network support and minimal yet critical application information. Enabling PD3 is a data processing unit or DPU, which allows (1) parsing user requests before they are processed by the compute server, (2) fetching data from remote memory on the shortest path, (3) offloading expensive RDMA and DMA operations from the host, and (4) incorporating application knowledge to faithfully predict cache misses and take actions accordingly. Designing PD3 requires reconciling DPU resource constraints and scaling requirements of cloud data systems, as well as achieving high efficiency with a myriad of performance optimizations. Our experimental results on real hardware, applications, and workloads show that with nominal compute-local memory, PD3 minimizes the performance gap between memory-disaggregated applications and their monolithic counterparts.

1 Introduction

Memory disaggregation, an architecture that splits monolithic compute servers into network-connected compute and memory pools, is a recent technological shift in cloud data centers that has fundamental implications on cloud applications [12, 13, 18, 23, 25, 33, 34, 45, 57]. Due to unprecedented resource elasticity and management flexibility, combined with ample data movement, the impact of this data center architecture on cloud data systems (e.g., databases and KV stores) is profound. In response, novel cloud data system designs have recently been introduced to work with disaggregated memory, from hash indexes [64], trees [37–39, 49, 50, 63], cache [32, 54], and shuffling [41] to offloaded OLAP operators [31, 59] and overall KV [28, 46] and database architectures [15, 51, 53, 60].

In memory-disaggregated systems, compute nodes use their local memory to cache frequently accessed data items, as illustrated in Figure 1. When the entire working set fits in

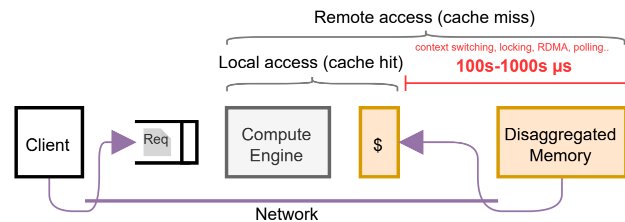
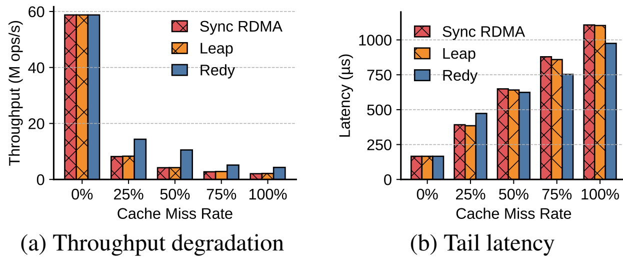


Figure 1: **Request execution with disaggregated memory.**

cache, the execution performance is identical to that of monolithic executions after warming up [58]. Cache misses, while expected when processing large-scale workloads with disaggregated memory, invoke out-of-core code paths that are much less efficient and more expensive, e.g., context switching [43], network communication [18], and response polling and processing [54], thereby leading to significant performance degradation [58]. This overhead is more pronounced for memory-disaggregated data systems, which often explicitly differentiate local and remote memory for more effective cache management and abstract the latter as secondary storage for easy integration. Compared to in-memory execution, accessing secondary storage performs extra work such as serialization and thread-safety and is thus more cumbersome. We empirically demonstrate the overhead of cache misses with a memory-disaggregated hash map that serves random lookup requests. We fix the local memory size to 1 GB and vary the hash map size and workload to control the cache miss rate. Detailed specifications for applications, hardware, and experimental setup can be found in Section 8. From Figure 2a, we observe throughput degrades from 59 million operations per second (ops/s) to 2 million ops/s. Cache misses also increase tail latency (p99) by up to 5× as shown in Figure 2b.

Existing data systems mitigate the overhead of memory disaggregation with three approaches. (1) *Overlapping compute and communication*, where remote memory accesses are executed asynchronously such that CPUs on the compute node can perform meaningful work before the requested data is retrieved. Examples of this approach include the asyn-



(a) Throughput degradation (b) Tail latency

Figure 2: **Cache misses lead to performance degradation.**

chronous device abstraction in Redy [54] and the lightweight context switch in AIFM [43]. Although this approach improves network utilization and overall execution efficiency (i.e., throughput), it does not improve time to response (i.e., latency), and its benefit highly depends on the workload. (2) *Compute offloading*, where memory-intensive operations are offloaded to the memory nodes for near data processing. Examples include operator pushdown in TELEPORT [59] and Farview [31], remote pointer chasing in CompuCache [55], and LSM-tree compaction offloading in dLSM [50]. These proposals effectively reduce data moved between compute and memory but are orthogonal to caching and are only effective for compute-light operations. Given the constrained computational power on memory nodes, most of the execution is expected to remain on compute nodes. Similarly to the previous approach, cache misses are still generated.

Finally, (3) *more effective caching*, where the data system *proactively* reduces cache misses by adopting optimized cache management strategies to improve the hit rate of the compute-local memory. Examples include the two-level LRU in LegoBase [60] that manages database pages in compute-local cache and remote buffer pool separately, and pinning small but frequently accessed items such as the index, directory, and metadata cache in Sherman [49], RACE [64], and FlexChain [51], respectively. These strategies to some extent incorporate workload characteristics and system-specific insights to reduce cache misses, especially when accessing critical data. Prefetching has also been exploited. In particular, Leap [40] is a prefetcher designed for memory disaggregation. It maximizes cache hit rate using the Boyer-Moore vote algorithm to detect the current major trend and filter transient accesses. Despite promising results presented, cache misses are "inevitable" with these caching solutions in cases of compulsory misses in cold starts and irregular access patterns. Indeed, Figure 2 shows that even with LRU, pinned metadata (remote address translation and hash collision resolution), and Leap-style prefetching, the cache miss rate remains high. In the prefetching case, false positive predictions bring useless items that pollute the cache and thus even degrade performance [40]. *Can we build a compute-local cache that addresses these problems, i.e., cold starts, irregular access patterns, and false positive mispredictions, to minimize the overhead of memory disaggregation due to cache misses?*

To answer this question, we design PD3, a memory disaggregation solution that augments the compute servers with a novel data prefetcher. Unlike a traditional prefetcher that runs on the compute server and predicts potential cache misses during request execution, PD3 places its prefetcher in the network, specifically, on the compute server's network interface controller, which allows the prefetcher to inspect data accessed by an incoming request and take actions before the request is processed by the execution engine on the host and possibly generates cache misses. Enabling the prefetcher are a family of emerging data center networking devices called Data Processing Units (DPUs), which are essentially System-on-a-Chip (SoC)-based SmartNICs. Recent work has explored the benefits of offloading file operations, shuffling, and database indexes to DPUs [26, 27, 35, 56, 62]. Our key insight is that a DPU, as a device that delivers network traffic, can not only detect cache misses before request execution but also fetch data from disaggregated memory using the fastest path (from NIC to remote memory). Together, early detection and fast resolution essentially enable *preventive prefetching:* cache misses and the associated cost are prevented from generation during execution.

Specific challenges of developing PD3 and technical contributions we make are as follows.

Determine, not predict, cache misses in advance. Traditional prefetchers learn data access patterns from execution and predict cache misses to maximize spatial and temporal locality [40]. As discussed, these prefetchers suffer from cold starts, irregular data access, and cache pollution. To determine cache misses, for each request received on the DPU, PD3 needs to know what data items will be accessed by the request and whether these items have been cached in host memory. The first piece of information varies between applications, and the latter requires visibility to the host memory. PD3 introduces two components on the DPU to address this challenge: a user-defined parser that translates each network request into a sequence of read/write operations over keyed items and a data structure that bookkeeps host-cached items and is opportunistically updated.

Prefetch disaggregated memory on the shortest path. Once cache misses are determined, the prefetching procedure should be initiated to read the missing items from remote memory. The shortest path to remote memory is directly issuing RDMA reads from the DPU. However, hijacking the connections between the host and remote memory nodes would violate RDMA's end-to-end semantics. To solve this dilemma, PD3 offloads the RDMA execution completely to the DPU such that remote memory access can be performed directly from the DPU. This offloading also effectively lowers host resource consumption.

Place prefetched items for fast application access. Once missing items are received on the DPU, they should be delivered to the host application for consumption. Achieving high data path efficiency for prefetched items is crucial yet challenging due to DPU-host communication and access con-

tention caused by high-performance applications. PD3 first reserves a chunk of host memory and registers it to the DPU driver for direct memory access (DMA). Prefetched items are asynchronously placed—with optimized DMA operations—in this buffer to shorten the data path. Further, to avoid out-of-core inefficiencies, PD3 provides a userspace interface for applications to access the prefetched items. The efficiency of this interface and its scalability to handle concurrent access minimizes the gap between cache-hit and prefetched code paths in high-performance data systems.

To summarize, this paper contributes PD3, a DPU-enabled prefetching framework for disaggregated memory. It determines cache misses before request execution and implements an efficient memory-to-compute data path. We present the architecture of PD3 (§4) and its detailed design (§5–§7). Evaluated with real workloads (§8), memory-disaggregated applications are up to an order of magnitude faster with PD3 than with existing solutions and close the performance gap to local memory with a nominal compute-local cache.

2 Background

Memory Disaggregation. Broadly, there are three forms of disaggregated memory abstraction at different levels: hardware, OS, and application. The Compute Express Link (CXL) [5] standard abstracts memory disaggregation behind traditional load/store instructions and provides coherent interconnect between compute and memory via extended PCIe. Despite increasing popularity, CXL-based memory disaggregation currently has limited deployment and its actual cost, performance characteristics, and suitability remain unclear [24, 36, 61]. In comparison, Remote Direct Memory Access (RDMA) has been a feasible solution to memory disaggregation [12, 13, 18, 23, 25, 31, 33, 34, 45, 49, 54, 57]. It has access to memory resources at large scale, enables application-specific optimizations, and is deployable in today's data centers. A line of RDMA-based memory disaggregation systems [25, 45] abstract remote memory behind virtual memory management in the kernel (e.g., page faults and swapping) and thus make disaggregation transparent for applications, providing backward compatibility. With this OS-level disaggregation, however, accessing remote memory introduces kernel overhead, and applications have no ability to distinguish local and remote memory. Application-level disaggregation, which provides userspace API for applications to explicitly manage compute-local cache and remote memory to bypass the kernel and enable optimizations as discussed in Section 1, has been more widely explored, especially for cloud data systems [18, 31, 43, 49, 54, 60]. PD3 is designed with this style of memory disaggregation.

Data Processing Units. Our proposal is enabled by DPUs. As an SoC, a DPU board provides the following resources: a high-speed network interface capable with 100s of Gpbs of bandwidth and support for Ethernet, InfiniBand and other protocols; power-efficient, software-programmable, multi-core CPUs that are tightly coupled with the rest of the SoC; on-board DDR memory for hosting the working sets of offloaded programs, which has limited size (e.g., 16-32 GB) compared to the host; a PCIe interface that provides efficient access to host memory and other PCIe devices. While DPUs provide high flexibility and efficiency for packet processing and moving data between the host, network, and peer PCIe devices (e.g., SSDs and GPUs), their constraints are obvious: the SoC cores are weaker than the host, and their core count and memory capacity is also significantly lower. Previous work explored using DPUs to offload KV [17, 42], file [30, 56], and database index [47] and shuffle [35] operations. In this work, we leverage DPUs' abilities to observe requests before execution on the host and perform fast RDMA communication to reduce the overhead of memory disaggregation.

3 System Requirements

To minimize the overhead of memory disaggregation, a system ideally avoids cache misses, i.e., a data item is readily available in the compute-local cache when it is accessed. To that end, an effective caching solution to run cloud data systems on compute servers meets the following requirements.

1. *Early access prediction,* such that sufficient time is given to prepare the data items that are not locally cached.
2. *High prediction accuracy,* such that the missing items that will be accessed in the short term will be prepared.
3. *Low false positive rate,* such that useless data items will not pollute the constrained compute-local memory.
4. *Fast preparation,* such that within limited time budget the missing data items are prepared before they are accessed to avoid execution stalls.
5. *Minimal application modification,* such that preparing missing items occurs transparently to existing applications, which then easily consume the prepared items.

Requirements (1)–(3) are especially challenging for cold starts and irregular accesses, and achieving the remaining two demands efficient interface and execution designs.

4 Architecture of PD3

Figure 3 shows the overall architecture of PD3. For applications, it offers remote memory as a set of byte-addressable regions and provides a familiar API to manage and access the regions: `Allocate` and `Deallocate` regions, `Read` and `Write` to arbitrary remote addresses, and `Poll` request completions, as proposed in prior work [12, 54]. Using this API, a data system (e.g., database or KV) executing on the compute server can use compute-local memory as cache and offload arbitrary state in its working set to remote memory. Although we heavily optimize remote memory operations with RDMA running on advanced networking hardware (200 Gbps bandwidth and single-digit μs latency), cache misses translate to network communication and are at least an order of magnitude slower than accessing the compute-local cache.

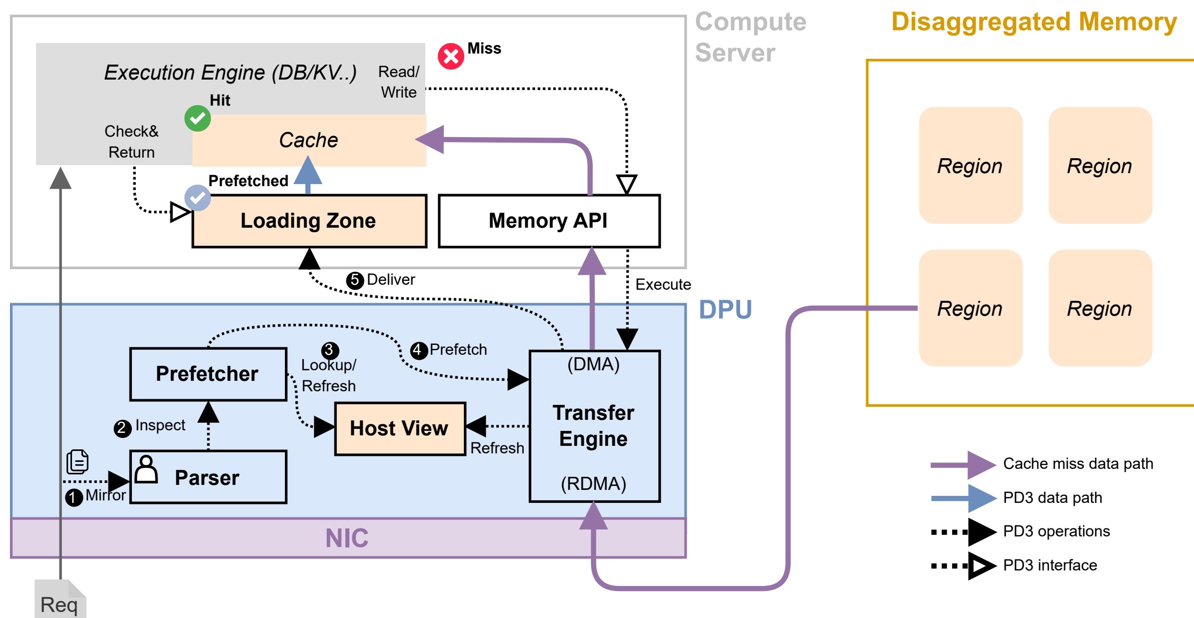


Figure 3: **PD3's architecture. When a network request arrives at the DPU, PD3 ❶ mirrors the request to the *Request Parser* that generates accessed data items, each of which is ❷ inspected by *Prefetcher*. Unlike traditional prefetchers that predict cache misses, Prefetcher determines them via ❸ *Host View*, which book-keeps host-cached items. Upon a positive lookup, Prefetcher then ❹ fetches the missed items from remote memory through the onboard *Transfer Engine*. When the requested items are received on the DPU, PD3 ❺ places them in *Loading Zone* for the application to consume.**

The key architectural contribution of PD3 is to avoid cache misses via DPU-enabled prefetching on the compute server. The DPU's unique position, its general programmability, and its hardware resources for network communication optimizations are essential to meet the aforementioned requirements.

For Requirement (1), PD3 inspects data access as soon as a request from the network arrives at the NIC before the host processes the request, an advantage offered by the DPU's position on the network path. Specifically, we adopt an efficient network engine on the DPU that mirrors every packet that carries user requests and arguments, which are immediately processed to inspect data items accessed by these requests. As the inspection does not modify application behavior, we push packet mirroring to the NIC hardware such that the original packets are forwarded to the host without any delay.

For Requirements (2) and (3), PD3's *Prefetcher* inspects what data items are accessed by each user request mirrored from the network and determines whether they are already cached on the host. It first allows users to specify a parser that takes as input a user request and its associated arguments and generates data items that will be accessed when the request is executed on the host. This user-defined *Request Parser* can accommodate various serialization protocols, e.g., gRPC, Redis serialization protocol (RESP), or any application-specific message format. To determine cache misses, we introduce a DPU data structure, called *Host View*, which maintains the view of the host cache, i.e., book-keeping the data items cached in the host memory. Host View allows for checking whether a data item accessed by a request is a cache miss without coordinating with the host application. This data structure is opportunistically maintained when new data items from the network or remote memory are loaded to the host cache and when existing items are evicted. With Request Parser and Host View, Prefetcher *guarantees no false positives*, i.e., it does not falsely fetch data items that will not be accessed by any request, thereby preventing cache pollutions.

For Requirement (4), once cache misses are determined, Prefetcher fetches the missing data items from remote memory using the shortest path, i.e., directly on the DPU. This path is enabled by *Transfer Engine*, which unifies data transfers between the compute host and the DPU via DMA and between the DPU and remote memory via RDMA. Hence, when Prefetcher determines a cache miss, it directly issues a read operation to the remote memory server with Transfer Engine, which executes the operation with optimized RDMA. The key benefit of offloading RDMA operations to the DPU is to allow remote memory access directly from the DPU without hijacking the end-to-end connections between compute and memory [18, 29]. Once the data item is returned, Transfer Engine forwards it to a PD3-reserved buffer in host memory.

Finally, for Requirement (5), the compute-local buffer reserved by PD3 is designed as an efficient host data structure, called *Loading Zone*. It lets applications consume prefetched data items with an intuitive API `CheckAndReturn`, which is invoked before entering the out-of-core code path.

These components proactively handle cache misses and

thus provide an efficient solution for data-intensive systems to utilize disaggregated memory with minimal overhead.

The next sections present in detail how PD3 overcomes the resource constraints on the DPU and performance bottlenecks, including low-latency and customizable request parsing (§5.1), a fast DPU data structure for Host View (§5.2), a unified data transfer engine (§6.1) that enables DPU-direct prefetching (§6.2), and the design of the `CheckAndReturn` API (§7.1) and Loading Zone (§7.2) for fast serving.

Assumption and Limitation: PD3 assumes the ability to parse data access (i.e., the unique identifiers of the objects in the working set and per-key operations in the requests) from network packets with Request Parser and small additional application-specific state. While this assumption does not universally hold, e.g., for complex SQL queries, it is applicable to many popular memory-disaggregated cloud-data processing systems: key-value stores [20], object cache [4,11], and block/page stores [14]. Compared to application-agnostic prefetching frameworks [40,43], PD3 is limited in its generality and requires application co-design. We plan to address this limitation in future work by also incorporating a generic fall-back prefetcher on the DPU.

5 Determining Cache Misses Early

The first step in the prefetching workflow is to parse user requests from network packets. Determining cache misses in a timely fashion requires efficient packet parsing and Host View querying. To maximize prefetching benefits without false positives, Host View must also be properly maintained.

5.1 Request Parser

User-defined Request Parsing. Clients serialize their requests into raw bytes as network packet payloads using standard protocols, e.g., protocol buffers [7] in gRPC, or system-specific serialization formats, e.g., Redis serialization protocol (RESP) [8] and PostgreSQL frontend/backend protocol [6]. To accommodate variety, PD3 provides the following interface for applications to customize request parsing.

$$\texttt{Parse}(\texttt{PktBuf}, \texttt{PktLen}) \rightarrow \texttt{List}\langle \texttt{Key}, \texttt{Op} \rangle$$

Specifically, the function takes as input the payload of a packet and its size and generates a list of keys of the accessed data items and their corresponding operations in their original order in the packet. Three operations are relevant to prefetching: `Read`, `Write`, and `Delete`, among which `Read` and `Delete` require accessing the keyed data items from the working set (`Read-Modify-Write` should be parsed as `Read`).

Low-latency Packet Mirroring. We leverage the embedded switch on the DPU NIC hardware to route packets at line rate. Packets that do not belong to the target application are filtered based on the flow ID, i.e., the 5-tuple of connections. Then, a potential solution to process packets of interest is to send packets of interest to the onboard CPU and forward them to the host when parsing is finished. However, detouring a packet to the DPU SoC can incur latency overhead. This is suboptimal for PD3, which targets high-performance applications backed by disaggregated memory. PD3 instead mirrors each packet of interest at the NIC and sends the copy to the user-defined parser on the DPU SoC. The payload of each packet is directly referenced by the parser without further copies. As such, low latency is achieved, and the original network path of the application is left untouched.

5.2 Host View

Host View is the core data structure that enables early in-network cache miss detection. Together with the parsed accesses, it allows PD3 to determine, not predict, cache misses. For each data access, i.e., `Read` or `Delete`, Host View should determine if the data item is cached on the host. Hence, it carries set membership semantics, but which keys to store has different implications as follows.

- Keys of the data items in the host cache.
- Keys of the data items missing in the host cache.

The former allows the set to store significantly fewer values as the majority of the application's working set resides in disaggregated memory. Compared to today's data stores, which can host billions of data items [52], DPU memory (16 GB–32 GB) is rather constrained. Hence, for better scalability, Host View is designed to bookkeep the keys of the data items cached on the host, realized as a hash table. However, it has further requirements on lookup performance and maintenance logic.

Performance Challenge. Modern data systems can process tens of millions of requests per second (§8). As each request parsed on the DPU needs to query Host View to determine if it is a cache miss, the lookup throughput should match the request processing rate of the application. Off-the-shelf hash tables struggle to exceed 10 million operations per second when handling hundreds of millions of keys. Figure 4 shows the throughput (in million operations/s) of different hash tables for 100 million 8-byte keys at 80% load: the two popular baselines, C++ STL `unordered_set` and Abseil hash set [3], achieve 3.7 and 6.3 million ops/s, respectively.

Optimization. Clients often batch small requests in a single network packet, e.g., the array of bulk strings in RESP [8]. Leveraging client-side batching, we merge multiple requests in a single Host View lookup. The multi-key lookup performance is optimized with the vectorized instructions and prefetching supported by the DPU cores, huge pages to reduce TLB misses, delayed hash computation and key search of the second bucket only when the key is not found in the first bucket to avoid unnecessary computation, and sharding the keys to scale to multiple DPU cores. We have incorporated the optimizations into a bucketized cuckoo hash table [22]. Figure 5 shows the pseudocode for a query to Host View with a batch of keys: Lines 2–4 apply a hash function to compute the first cuckoo bucket for each key and prefetch the slots of that bucket. Lines 5–6 search the slots in each bucket efficiently with SIMD instructions. Finally, Lines 7–13 repeat

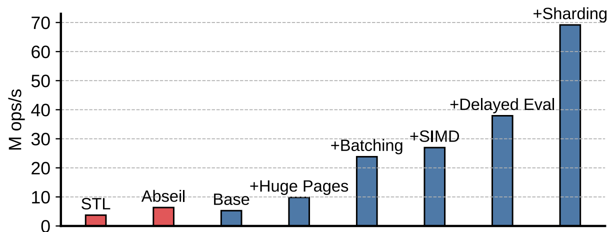


Figure 4: **Host View performance optimizations.**

```
1  FindBatched(KeyVec): #batched lookup
2    for k in range(0, KeyVec.size()):
3      bucket1Vec[k] = HashFunc1(KeyVec[k]) % BUCKETS
4      _PREFETCH(&bucket1Vec[k])
5    for k in range(0, KeyVec.size()): #search buckets
6      ResultVec[k] = FindSIMD(bucket1Vec[k],KeyVec[k])
7    for k in range(0, KeyVec.size()):
8      if ResultVec[k].IsNull(): #delayed hash
9        bucket2Vec[k] = HashFunc2(KeyVec[k]) % BUCKETS
10       _PREFETCH(&bucket2Vec[k])
11   for k in range(0, KeyVec.size()):
12     if ResultVec[k].IsNull(): #delayed search
13       ResultVec[k] = FindSIMD(bucket2Vec[k],KeyVec[k])
14   return ResultVec

15 FindSIMD(Slots, Key): #search a bucket via SIMD
16   foundVec = _VEC_COMP(Slots, Key) #SIMD compare
17   return _VEC_MOVEMASK(foundVec) #SIMD movemask
```

Figure 5: **Querying Host View.**

these steps for the second bucket (delayed evaluation) for any key not found in its first bucket. Lines 15–17 search a given bucket using a SIMD `compare` for all-slot comparison and then apply a SIMD `movemask` to generate the search result.

Figure 4 shows the impact of these optimizations: although the basic cuckoo hash set is not faster than the Abseil's, with huge pages it achieves 9.8 million ops/s. Batching multiple keys significantly increases the throughput to 23.4 million ops/s. Applying SIMD instructions boosts the throughput to 27 million ops/s. Delayed hash and search further brings a 40% improvement. Finally, sharding Host View with two threads nearly doubles the throughput to 70 million ops/s.

Maintenance. A poorly-maintained Host View may cause false positives (i.e., prefetch an already-cached data item) or false negatives (i.e., miss the prefetching opportunity for a missing data item). For example, if a blind write request on data item D is successfully executed by the host but is not reflected in Host View, a later read request on D will be falsely treated as a cache miss by PD3's Prefetcher.

Table 1 lists all possible scenarios where the data items in the host cache can change. Specifically, new data items can be added to the host cache from three sources: (1) write operations from clients, (2) read responses from remote memory, and (3) silently generated by the host. Missing any of these events in Host View can potentially cause false positives. PD3 reflects writes from clients by adding the keys of parsed write operations to Host View, and the reads by adding to Host View the keys in read responses from remote memory (detailed in §6.1). Data items silently generated by the host are not reflected in Host View, but fortunately, they are also absent in remote memory. A prefetching request for such a nonexistent key will return an error. Hence, missing the last change does not result in cache pollution on the host.

PD3 may generate false negatives. As Table 1 shows, existing data items cached on the host can be deleted in three scenarios: (1) delete operations from clients, (2) write requests to remote memory for flushing, and (3) silently dropped by the host (normally when clean data items are replaced). In the first two scenarios, the keys of parsed delete operations and the keys of write requests are removed from Host View. Similar to silent additions, silent data deletions on the host are not reflected in Host View. However, false negatives have no adverse effect on the application performance, as cache misses will be handled in the host application by reading the missing data items from remote memory, using the normal cache miss path—future accesses to the same data item will not result in false negatives.

Table 1: **Possible changes to the host cache.**

Change	Source	Impact	Reflected
Addition	Writes from clients	False Positive	Yes
	Reads from memory	False Positive	Yes
	Generated by the host	None	No
Deletion	Deletes from clients	False Negative	Yes
	Flushes to memory	False Negative	Yes
	Dropped by the host	False Negative	No

Failure Discussion. Failures are a special case for Host View. We expect in most failures, for example, power outage, there is fate sharing between the host and the DPU. In cases where the DPU fails but the host survives, which is essentially a network partition, restarting the compute server synchronizes Host View and the host cache. In remaining cases where the host fails but the DPU survives, existing keys in Host View can result in false negatives. As discussed, false negatives do not adversely impact application performance. We leave handling such cases to future work.

6 Shortest-path Prefetching

Transfer Engine runs on the DPU and unifies DMA and RDMA to transfer data between the host, the DPU, and remote memory. Three key benefits are enabled. (1) By accessing host memory with DPU-issued DMA, we can eliminate host resource consumption for host-DPU communication. (2) By offloading RDMA operations to the DPU to access remote memory, we can eliminate host resource consumption for compute-memory communication. (3) By enabling direct remote memory access on the DPU, we can pave the shortest path to prefetch missing data items. This section details Transfer Engine and PD3's shortest-path prefetching workflow.

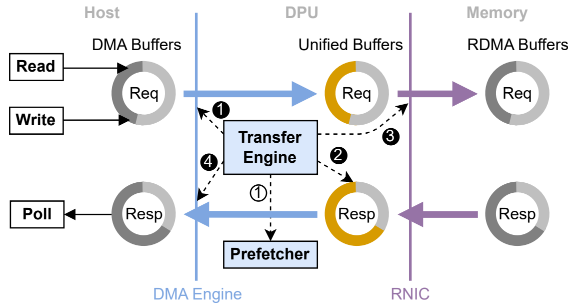


Figure 6: **Unified data transfers with PD3's Transfer Engine eliminate extra data copies on the DPU, enable DPU-direct remote memory access, and facilitate batching.**

6.1 DPU-offloaded Data Transfers

Unifying DMA and RDMA. DPUs have unique advantages for data movement optimizations. Specifically, a DPU's hardware DMA engine allows for direct access to host memory and other PCIe devices, and its RDMA interface (RNIC) allows for direct access to remote memory. PD3's Transfer Engine leverages these and introduces *unified buffers* to achieve zero-copy execution to avoid data copies in bridging these two protocols, as shown in Figure 6. Essentially, a unified buffer is a chunk of DPU memory accessible by both the DMA engine and the RNIC and can serve as the source or destination for both DMA and RDMA. When Transfer Engine starts, it allocates two chunks of DPU memory and registers them to the DMA engine and RNIC. At runtime, the buffers are used as rings to exchange data transfer requests and responses. Batching naturally occurs in the ring design as detailed shortly and thus does not sacrifice latency for high throughput.

Transfer Engine uses a unified buffer as the request buffer, which is the destination of DMA reads to accept remote memory access requests from the host and the source of DPU-issued RDMA writes to send the requests to the remote memory server. Conversely, the other buffer is used as the response buffer, which is the destination of remotely-issued RDMA writes to receive remote memory access responses and the source of DMA writes to transfer the responses to the host.

Host-DPU data transfers with DMA. PD3 provides host applications with a light-weight library to invoke its memory API to access remote regions, i.e., the `Read` and `Write` functions. When the API is invoked, a remote memory access request in the following format is generated:

2b	14b	2B	4B	4B	*Size* B (write)
Op	ID	Region	Offset	Size	Data

Specifically, a request consists of a 12-byte compact header that specifies and identifies the operation: a 2-bit operation code that differentiates different requests (read, write, or prefetching (§6.2)), a 14-bit request ID [1], 2-byte remote memory region ID and 4-byte offset that specify remote address, and finally a 4-byte size indicating the size of the request. In cases of write requests, the write data immediately follows the header. The library internally accumulates requests from multiple application threads in a single lockless ring buffer, thereby creating a batching effect. Transfer Engine polls the buffer and issues a DMA read to move all available requests to the unified request buffer (Step ❶ in Figure 6).

[1] PD3 issues no more than 16 thousand in-flight requests.

The library allocates another buffer for receiving responses:

2b	14b	4B	*Size* B (read)
Op	ID	Size	Data

The operation code and request ID are carried over, followed by the data size successfully accessed and the read data. When remote memory access responses are received, Transfer Engine issues a DMA write to forward all available responses on the unified response buffer to the buffer (❹). The invocation of the `Poll` function inspects the buffer and completes the respective requests if responses are available.

DPU-memory data transfers with RDMA. When a batch of remote memory access requests is polled from the host, Transfer Engine first calculates the response size (i.e., $6 \times |AllRequests| + \sum_{r \in ReadRequests} r.Size$ bytes) for these requests and reserves sufficient space in the unified response buffer (❷). This space serves as the destination of the RDMA write issued by the memory server to deliver the responses. Then, Transfer Engine constructs an RDMA write with the unified request buffer as the source to send out the requests (❸). Upon receiving these requests, the memory server executes reads and writes accordingly on the local regions. In addition, the server builds a key-to-address mapping using the user-defined request parser (§5.1) for each write request, which is used to execute prefetching requests (§6.2).

Zero-copy execution. The above data transfer procedures are carefully designed to avoid data copies on the DPU. In addition to the unified buffers that eliminate copies between DMA and RDMA buffers, pre-allocating RDMA response space also avoids extra response copies. In fact, Transfer Engine only issues DMA and RDMA commands to *let the corresponding hardware move data*, thereby achieving high transfer speeds with a single DPU SoC core. The serialized execution on a single core also provides read-after-write consistency when a prefetching request is issued immediately after a fresh removal from Host View.

Updating Host View. As discussed in Section 5.2, Transfer Engine should add to Host View the keys of new data items read from remote memory and remove the keys of data items flushed to remote memory to minimize false negatives and eliminate false positives for prefetching. Transfer Engine updates Host View based on the rules in Table 2. Specifically, after receiving requests from the host (Step ❶ in Figure 6),

Table 2: **Action table for updating Host View.**

Action	Operation	Occasion
`Add`	`Read`	Before the DMA write ❹
`Del`	`Write`	After the DMA read ❶

it scans the requests, applies the user-defined request parser to generate keys from the write data, and removes them from Host View; before forwarding the responses received from the remote memory server to the host (❹), it scans the responses and populates Host View with the read keys. Note that the delayed additions to Host View could trigger duplicate prefetching for the same key, which PD3 prevents by bookkeeping the in-flight prefetching requests.

Performance of Transfer Engine. To measure the speed and host CPU saving of Transfer Engine, we transfer 64 B records from the host to the DPU and from the host to the remote memory server and compare it with host-issued RDMA to transfer data to the DPU and remote memory. Transfer Engine achieves 1.5× higher throughput with zero host CPU consumption than host-issued RDMA with 16 CPU cores. Detailed results are reported in Appendix A.

6.2 DPU-direct Prefetching

A key benefit of Transfer Engine is to enable the shortest path to prefetch missing data items from remote memory. To do so, Transfer Engine takes an additional step in its workflow to poll keys from Prefetcher (Step ① in Figure 6). For each key, it inserts the following request into the unified request buffer:

2b	14b	*K* B
Op	ID	Key

The key size K can be configured. Steps ❷❸ are then followed to send the requests to the remote memory server. For each request, the server queries its key-to-address mapping to read the data item and generates the following response:

2b	14b	31b	1b	*K* B	*Size* B
Op	ID	Size	State	Key	Data

The first two bytes are carried over from the request. A 4-byte size indicates the size of the data item, with the last bit piggybacking a state (thus we support data item sizes up to 2 GB), followed by the K-byte key. Each data item has two possible states: PRODUCED and CONSUMED, and set by the server as PRODUCED. These states are used by Loading Zone (§7.2). After executing all the requests, the server RDMA-writes the responses to the unified response buffer in Transfer Engine. Step ❹, serving prefetched data items to applications has more stringent performance requirements, which PD3 addresses with Loading Zone.

To show the benefit of directly accessing remote memory for prefetching on the DPU, we compare it to an alternative path where each prefetching request is first forwarded to the host and then sent to the memory server. Compared to PD3 (DPU/NIC → memory → DPU/NIC → host), this path incurs additional DMA roundtrips (DPU/NIC → host → DPU/NIC → memory → DPU/NIC → host). PD3's prefetching path is up to 24% faster, in addition to saving host CPU cycles.

7 Efficient Serving

The last mile of the prefetching workflow of PD3 is to serve the prefetched data items to the host application (e.g., a KV service or a DBMS). A potential solution is to place the data items behind the memory API and intercept the read call. However, the memory API is already in the out-of-core code path. A potential efficient solution is to co-design this step with the application's buffer manager such that the latter makes space in the host cache for the incoming data items from the DPU and incorporates additional caching policies to manage these items. However, this violates Requirement (5). PD3 achieves generality and efficiency by introducing an in-memory interface and a data structure that supports the interface with minimal contention.

7.1 Interface

API. To consume prefetched items, applications call:

$$\texttt{CheckAndReturn}(\texttt{Key}, \texttt{DestBuf}) \rightarrow \texttt{Size}$$

If the requested key is found in Loading Zone, the data item is copied to `DestBuf` and returns the size of the data item; otherwise, 0 is returned. This function is non-blocking, i.e., the result simply depends on what data items have been prepared by PD3 and will not block for in-flight prefetching requests. This non-blocking design aligns with pipelined request execution in high-performance data systems.

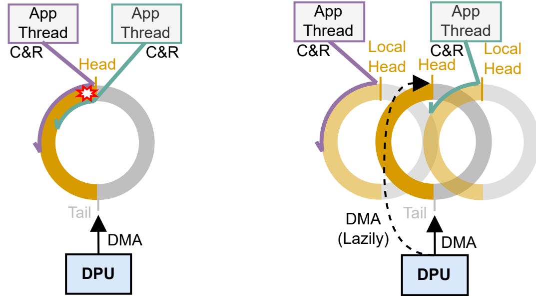
Target Usage. This easy-to-use function is supposed to be invoked *before* entering the out-of-core code path to avoid potential overhead and to retrieve, opportunistically, a missing data item. As we show shortly, the performance of this function is optimized to match the in-memory execution speed.

7.2 Loading Zone

Loading Zone is accessed when `CheckAndReturn` is invoked.

Organization. The host buffer reserved by PD3 is organized as a ring buffer to minimize the coordination between the DPU and the host. The DPU (i.e., Transfer Engine) acts as the producer that writes prefetched data items via DMA. Application threads that call `CheckAndReturn` act as consumers that look up requested keys and, if found, remove the data items. As regular ring buffers, it uses two pointers `Head` and `Tail` to coordinate the producer and consumers. The DPU producer writes prefetched data items contiguously into the buffer from `Tail` and updates the pointer when finished. Each consumer searches its missing key among the data items produced by the DPU from `Head` to `Tail`.

Contention. Consumers in existing ring buffers proactively update the head pointer when a data item is consumed [1, 2, 10, 21, 48, 56]. This creates high contention when many threads concurrently consume data from the buffer. As shown in Figure 7a, when `CheckAndReturn` is invoked, the consumer peaks the available data items in the buffer sequentially from `Head`. If the current data item matches the requested key, it is copied to `DestBuf` and then deleted. Additionally, it advances `Head` to the last unconsumed data item. There are two spots for contention in this process: (1) same-key contention (SKC): if multiple consumers have the same missing key found in the buffer, they



(a) Contention between threads (b) PD3 replicates `Head`

Figure 7: **Loading Zone design.**

```
1  CheckAndReturn(Key, DestBuf):
2    LocalHead = Head #replicate Head to eliminate HC
3    while LocalHead != Tail:
4      D = (PD3_Header)LocalHead
5      if Compare(D.Key, Key) == true:
6        Copy(DestBuf, D.DataItem, D.Size)
7        if D.State == PRODUCED:
8          CAS(D.State,PRODUCED,CONSUMED) #reduced SKC
9          return D.Size
10     LocalHead += sizeof(PD3_Header) + D.Size
11   return 0
```

Figure 9: **Consuming Loading Zone.**

must coordinate to consume and remove the data item, and (2) head contention (HC): if multiple consumers find their keys, they must coordinate to update `Head`. Figure 8 shows the performance achieved by Loading Zone with varying numbers of application threads invoking `CheckAndReturn`. We observe that although lock-free design with atomic operations can reduce the contention overhead (the throughput is increased up to 40× by replacing spinlocks with compare-and-swap instructions), the throughput is capped at 5-10 million invocations/s. Although this level of performance suffices for storage and network I/O, which is the target scenario of existing ring buffers (e.g., those in FaRM [21], DFI [48], and Redy [54]), PD3 seeks to close the performance gap between disaggregated and local memory. Therefore, Loading Zone is expected to match in-memory execution performance.

PD3's Solution. PD3 utilizes the prefetching response header, i.e., for each data item *D* in the buffer, the header includes three essential attributes: *Size* that denotes the length of *D*, *State* that indicates if *D* has been consumed, and *Key* that maintains the key of *D* for comparison. As shown in Figure 7b, upon an invocation of `CheckAndReturn`, a consumer replicates `Head` as its `LocalHead` and advances it to visit all data items until `Tail`. For each data item *D*, it first compares *D.Key* with `Key`. If it is a match, it copies the data item to `DestBuf`, and then checks *D.State* before returning *D.Size*. If *D* is PRODUCED, it executes a compare-and-swap (CAS) instruction to toggle the state to CONSUMED. If *D.Key* does not match `Key`, the consumer advances the `LocalHead` to the next data item by $6+K+D.Size$ bytes. If `LocalHead` reaches `Tail`, 0 is returned. Figure 9 shows the pseudocode for this process, which minimizes SKC and entirely eliminates HC. The only contention is the CAS operation on a binary variable.

The DPU advances `Tail` via a DMA write after DMA-writing the prefetched data items. Once `Tail` reaches `Head`, the DPU performs cleaning to advance `Head`: if *D.State* is CONSUMED for the item *D* at the current `Head`, `Head` is advanced past that item; otherwise, the cleaning stops.

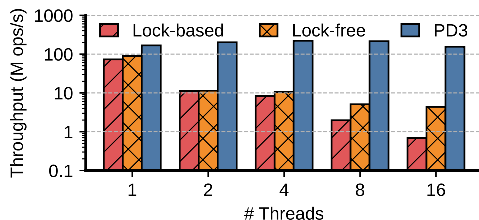


Figure 8: **Concurrent performance of Loading Zone.**

Figure 8 shows that PD3's Loading Zone achieves ∼150 million invocations/s even when there are 16 threads concurrently calling `CheckAndReturn`—224× *and* 35× *faster than the locking and lock-free baselines, respectively*.

8 Evaluation

We answer the following questions about PD3:

- How well can PD3 improve the performance of memory-disaggregated applications with constrained compute-local cache compared to existing approaches?
- What is the contribution of prefetching?
- What are the benefits of saving host resources?
- How sensitive is PD3 to the compute-local cache size?

8.1 Methodology

Testbed. Our testbed cluster is as follows.

- **DPU.** We use NVIDIA BlueField-3 (BF-3) as the DPU platform. It consists of a 200 Gbps network interface, 8 ARMv8 A78 cores, and 32 GB DDR4 memory.
- **Server Host.** Each of our server host nodes consists of an AMD EPYC 9254 CPU (24 cores @2.90 GHz), and 128 GB DDR5 memory, running Ubuntu 22.04.
- **Network.** We use a server equipped with the BF-3 DPU as the compute node. The client and the disaggregated memory are hosted on other nodes, which are connected to the compute node with 200 Gbps ConnectX links.

Prototype. PD3 has been implemented with ∼15,000 lines of C++ code spanning the DPU, host of the compute node, and remote memory node. On the DPU, packets are filtered and mirrored using the `Flow` engine in the NVDIA DOCA SDK, and the mirrored packets are processed by a DPDK program incorporating the user-defined request parser. The DMA and RDMA operations are implemented via DOCA `DMA` engine and `ibverbs`. On the compute server, a light-weight library provides access to the memory API and `CheckAndReturn`. It also implements the response buffers and Loading Zone by allocating memory and registering it for DMA. The resource usage of the system is summarized below.

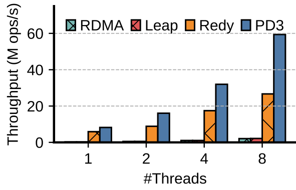


Figure 10: **Throughput of hash map (YCSB, uniform).**

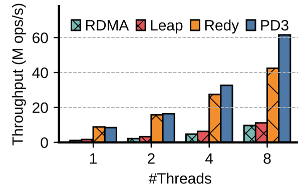


Figure 11: **Throughput of hash map (YCSB, Zipfian).**

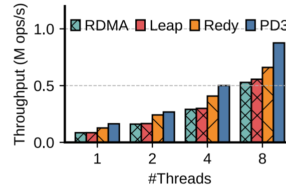


Figure 12: **Throughput of the page server (random access).**

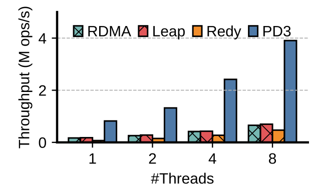


Figure 13: **Throughput of Garnet (RESP benchmark).**

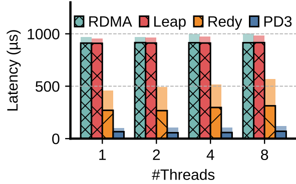


Figure 14: **Latency of hash map (YCSB, uniform).**

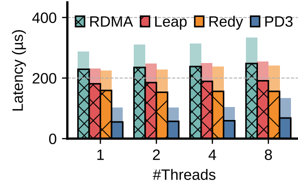


Figure 15: **Latency of hash map (YCSB, Zipfian).**

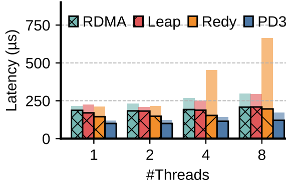


Figure 16: **Latency of the page server (random access).**

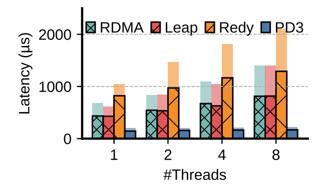


Figure 17: **Latency of Garnet (RESP benchmark).**

- On the DPU, four cores are utilized: ×1 for DPDK, ×1 for Transfer Engine, and ×2 for sharding Host View. The memory footprint of Host View is 100 million keys.
- No CPU cores are consumed on the compute server. 128 MB is sufficient for all PD3-managed buffers (§8.5).
- The memory server uses one CPU core to setup local memory regions and process requests.

The source code for PD3 is available at `https://github.com/fardatalab/PD3`.

Applications and Workloads. We evaluate three applications that can benefit from disaggregated memory: (1) an open-address hash map with linear probing that consists of key-value records and scales by offloading most records to remote memory, (2) a database page server that services page read requests and stores most of the managed pages in remote memory, and (3) Garnet [4], a production KV cache service from Microsoft, backed by a hybrid log [16] that spills to remote memory when local memory is full. Integrating these applications with PD3 is intuitive: they use the memory API to access remote memory and, to enable prefetching, invoke `CheckAndReturn` before entering their out-of-core code paths. The user-defined request parsers are provided for each application: in-house formats for the hash map and the page server, and a parser for Garnet that parses client requests with RESP [8] and the system's internal hybrid log requests.

We run YCSB [19] (1 billion keys, mixing 95% reads and 5% writes) for the hash map, irregular random page read requests for the page server, and the RESP benchmark [9] (425 million keys) for Garnet where keys are uniformly accessed. The YCSB and RESP benchmarks involve a 8 B key and a 8 B value, and for the page server, each page has a 8 B id and 4 KB data, and the server stores 10 million pages. Unless otherwise stated, the working set of each workload fits in the memory server, and the compute server allocates 1 GB local memory as cache that holds a fraction of it.

Compared Approaches. We validate PD3's performance by comparing it with three memory disaggregation solutions:

- Host-issued synchronous RDMA reads and writes.
- Redy [54], a recent system that abstracts remote memory as a byte-addressable device and provides an asynchronous API. Its RDMA data path is heavily optimized.
- Leap [40], a recent prediction-based prefetcher for disaggregated memory using Boyer-Moore voting to detect access patterns. We implement it in userspace, and when cache misses occur, it falls back to synchronous RDMA.

8.2 PD3 Improves Overall Performance

We first evaluate how PD3 improves the overall performance of memory-disaggregated applications compared to current solutions. Figures 10 and 11 show the throughput of the hash map with the YCSB uniform and Zipfian (with skewness $\theta = 0.99$) workloads. This experiment evaluates the performance for accessing small data items. Between existing approaches, synchronous RDMA achieves the lowest performance, barely exceeding 2 million ops/s and 9 million ops/s for uniform and skewed access patterns, respectively. Leap's prefetching brings 15% improvement to the skewed workload but has no effect on the uniform workload where accesses are unpredictable. Redy provides asynchronous, batched, and pipelined remote memory access and is up to 9.5× faster than synchronous RDMA. Despite significant improvement, remote memory access remains an expensive operation, as significant throughput degradation can be observed from the skewed workload to the uniform workload. PD3 instead determines cache misses and leverages a fast prefetching path to prepare missing items before requests are executed. The hash map with PD3 achieves 26× and 2.2× higher throughput than RDMA and Redy, respectively, for the uniform workload.

Figure 12 shows the throughput of the page server, where accesses to remote memory are significantly larger. This application represents a challenging case for disaggregated memory, where the network bandwidth becomes the bottleneck. PD3 remains beneficial due to its advanced pipelining—data

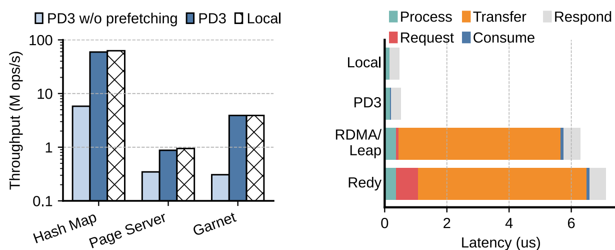


Figure 18: **Comparing PD3 w/ and w/o prefetching.**

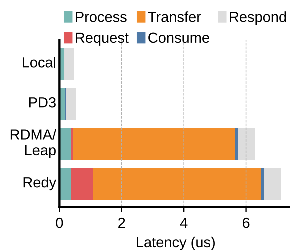


Figure 19: **Garnet request execution breakdown.**

starts flowing from remote memory before a page request is processed by the host. It achieves 65% and 33% higher throughput than synchronous RDMA and Redy.

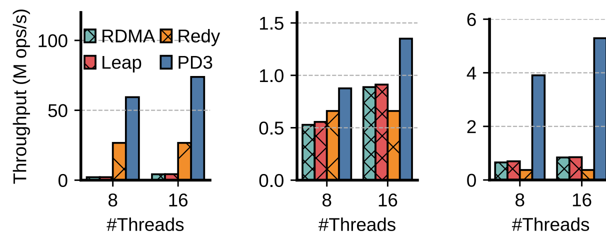
We finally evaluate a production key-value cache service. Figure 13 shows the throughput of Garnet with the RESP benchmark. Perhaps surprisingly, RDMA optimizations, such as batching and pipelining in Redy, negatively impact Garnet's performance. This is because, to comply with the RESP protocol, Garnet disallows more than one request to be active on a given thread. Therefore, Garnet achieves 1.4× higher throughput with synchronous RDMA or Leap than with Redy. Since PD3 optimizes RDMA data transfers in the background (DPU) and provides the efficient `CheckAndReturn` API in the foreground (host) targeting in-memory usage, Garnet can enjoy its performance benefits like other applications. Its throughput with PD3 is 6× higher than that with synchronous RDMA/Leap and 8.5× higher than that with Redy.

Figures 14–17 report the median and tail (p99) latencies of the hash map (with uniform and skewed workloads), the page server, and Garnet when achieving the throughput in Figures 10–13. In the hash map, Redy's RDMA optimizations are effective and do not incur overhead. Its latency is up to 3.5× lower than that of synchronous RDMA and Leap. However, in the page server, where requests are large, and Garnet, where asychrony is disabled in the application, its optimizations backfire and incur up to 2.2× higher latency than synchronous RDMA and Leap. PD3 converts remote memory access to local operations. It consistently achieves lower latency than other approaches across all the applications, ranging from 1.5× to 11× reductions for median latency and from 1.8× to 10× reductions for tail latency.

Summary: PD3 enables significant improvements (up to an order of magnitude) to the throughput and latency of memory-disaggregated applications compared to existing solutions.

8.3 Prefetching Is Essential

We next evaluate the contribution of prefetching to PD3's performance improvement. Specifically, we measure the changes in application performance with prefetching enabled and disabled in PD3. When prefetching is disabled, applications fall back to calling the memory API. We also compare PD3 with local memory by increasing the compute-local cache size to



(a) Hash map (uniform) (b) Page server (c) Garnet

Figure 20: **Scaling applications with saved host CPU cores.**

hold the entire working set. Figure 18 shows the throughput of the hash map (uniform workload), the page server, and Garnet. We first observe that, when prefetching is disabled, PD3 does not outperform existing approaches. Applications are up to 12.7× lower than with the full PD3. This result confirms that the primary improvement enabled by PD3 is due to its prefetching design, rather than RDMA offloading to the network [18]. This experiment also shows PD3 closes the performance gap between disaggregated and remote memory: their performance differences are under 10%.

Figure 19 further breaks down Garnet request execution to multiple stages: (1) `process`—the time to check the requested item in the local cache and, in case of a cache miss, invoke the disaggregated memory API, (2) `request`—the time to generate a remote memory access request to fetch the missing item, (3) `transfer`—the time to move the item from remote memory to the compute server, (4) `consume`—the time to consume the fetched item, and (5) `respond`—the time to generate the final response for the client. In cases of cache hits, only `process` and `respond` are involved, which take 500 *ns*; PD3 further involves `consume`, which merely incurs 38 *ns*. Existing disaggregated memory solutions (synchronous RDMA, Leap, and Redy) inevitably trigger `request` and `transfer`, which are the most time-consuming stages. They take 5.3 μs in synchronous RDMA and Leap and 6.1 μs in Redy due to additional batching overhead in `request`.

Summary: PD3 closes the performance gap between local and disaggregated memory with its DPU-enabled prefetching.

8.4 Saved Host CPUs Are Useful

PD3's prefetching workflow executes on the DPU and consumes zero cores on the compute server. In comparison, Redy by default spawns an I/O thread per application thread to avoid shuffling between application and I/O threads and thus doubles the core utilization. Figure 20 shows PD3's benefits for saving host CPU cores. Specifically, Redy's throughput for each application saturates at 8 application cores. In comparison, PD3 can further allocate additional CPU cores on the host to applications. With 16 application threads, PD3 increases the throughput of the hash map (uniform workload), the page server, and Garnet to 73 million ops/s (+30%), 1.4 million ops/s (+54%), and 5.3 million ops/s (+35%), respectively, compared to 8 application threads.

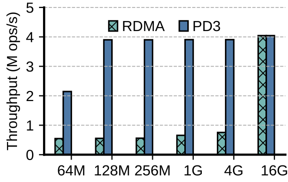


Figure 21: **Throughput with varying local memory sizes.**

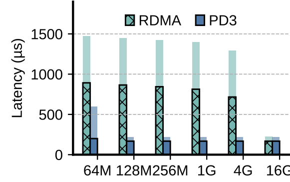


Figure 22: **Latency with varying local memory sizes.**

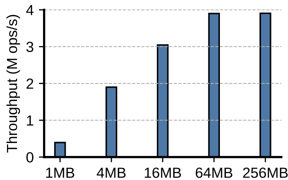


Figure 23: **Throughput with varying loading zone sizes.**

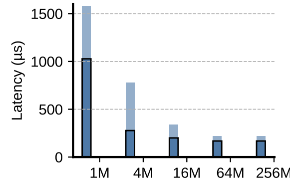


Figure 24: **Latency with varying loading zone sizes.**

Summary: CPU cores on the compute server saved by PD3 can be used by applications to further improve performance.

8.5 PD3 Facilitates Memory Disaggregation

The final aspect we investigate is how much memory PD3 requires on the compute server to close the performance gap between disaggregated and local memory. We run Garnet (similar observations apply to other applications) on the compute server and vary the local cache size from small (64 MB) to large (16 GB, which suffices to cache the entire working set). Figures 21 and 22 show the changes in throughput and latency (median and p99 tail), respectively. We use synchronous RDMA as a reference (Leap and Redy behave similarly as shown in Figure 2). We first observe that with only 128 MB compute-local cache—under 1% of the database size, Garnet achieves the same performance with PD3 as with local memory. In comparison, with existing disaggregated memory solutions, only when the whole database is cached on the compute server does Garnet eliminate cache misses and achieve the same performance as local memory.

We observe significantly lower performance with PD3 when further reducing compute-local cache size to 64 MB. To dissect the degradation, we vary the size of Loading Zone from 1 MB to 256 MB and observe that, as shown in Figures 23 and 24, the throughput deteriorates and the latency increases when the size is below 64 MB. This is because some prefetched items are replaced by more recent prefetches before they are consumed by the application. We believe that this fundamental trade-off between performance and the degree of disaggregation is inevitable, and PD3 drastically decreases the minimum compute-local memory required to avoid application performance regression in disaggregated memory. Even when the compute-local cache is below 128 MB, PD3 remains faster than the existing approaches.

Summary: PD3 facilitates memory disaggregation by minimizing the memory footprint on compute servers.

9 Related Work

Memory-disaggregated data systems. Disaggregated memory (DM) has been extensively explored in data system design. Li et al. [32] integrates remote memory into DBMSs via RDMA. PolarDB Serverless [15] proposes a database design that disaggregates computation, memory, and storage. FlexChain [51] is a permissioned blockchain on disaggregated computation and memory nodes. Motor [53] addresses the concurrency and performance challenges of distributed transactions in DM with MVCC. RDMA tree [63] explores and evaluates the different design spaces of distributed tree-based index for memory-disaggregated databases. Sherman [49] builds a write-optimized B+Tree index on DM. DEX [37] is B+Tree index for memory disaggregation with improved scalability. CHIME [38] is a hybrid index for DM that combines B+ tress with hopscotch hashing. dLSM [50] offloads LSM-tree compaction to DM. SMART [39] constructs a radix index for DM with fine-grained concurrency control. FUSEE [46] designs a client-centric protocol to replicate indices for KV. RACE [64] proposes a new hashing scheme to address weak compute power on memory nodes. PD3 provides a solution to minimize cache misses and thus the overhead of DM.
DPU-offloading for data systems. Several prior works applied DPU-offloading to improve data systems. Gimbal [42] leverages SmartNICs to implement efficient multi-tenancy for disaggregated storage. Xenic [44] accelerates distributed transactional systems using SmartNICs by partially offloading concurrency control. LineFS [30] offloads distributed file system operations to a SmartNIC to reduce host CPU overhead. DDS [56], part of DPDPU [26], offloads remote storage reads to DPU to shorten disaggregated storage latency and save server host CPU cycles. SmartShuffle [35] offloads the shuffle operation in distributed analytics to SmartNICs. PD3 differs from these works in its use of DPUs for prefetching purposes to optimize the efficiency of disaggregated memory.

10 Conclusion

PD3 is a solution to the overhead of memory disaggregation. It adopts a user-defined request parser and a DPU data structure to determine cache misses before requests are executed. A data transfer engine bridges the host, the DPU, and remote memory to facilitate efficient data prefetching. To serve prefetched data items, it introduces a fast host data structure that minimizes host thread contention. PD3's benefits have been empirically validated with three realistic applications.

Acknowledgments

We thank our shepherd Vasiliki Kalavri and the anonymous reviewers for their thoughtful feedback. This work was funded in part by Natural Sciences and Engineering Research Council of Canada (NSERC) RGPIN-2023-04834 and DGECR-2023-00308. Jialin Li was supported by the Singapore Ministry of Education, under Academic Research Fund Tier 2 grant MOE-T2EP20222-0016.

References

[1] Lockless ring buffer design, 2009.

[2] Ring library - data plane development kit, 2014.

[3] Abseil common libraries (c++), 2025.

[4] Garnet: A high-performance cache-store from Microsoft Research, 2025.

[5] Homepage - compute express link, 2025.

[6] PostgreSQL Documentation: Message Formats, 2025.

[7] Protocol Buffers Documentation, 2025.

[8] Redis Protocol specification, 2025.

[9] The resp.benchmark tool, 2025.

[10] Unraveling ebpf ring buffers, 2025.

[11] Redis: The Real-time Data Platform, 2026.

[12] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 775–787. USENIX Association, 2018.

[13] 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.

[14] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1743–1756, 2019.

[15] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. Polardb serverless: A cloud native database for disaggregated data centers. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2477–2489. ACM, 2021.

[16] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. FASTER: A concurrent key-value store with in-place updates. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 275–290. ACM, 2018.

[17] Hung-Hsin Chen, Chih-Hao Chang, and Shih-Hao Hung. hkvs: a framework for designing a high throughput heterogeneous key-value store with smartnic and RDMA. In Peng Li, Junyoung Heo, and Tomás Cerný, editors, *Proceedings of the Conference on Research in Adaptive and Convergent Systems, RACS 2022, Virtual Event, Japan, October 3-6, 2022*, pages 99–106. ACM, 2022.

[18] Xinyi Chen, Liangcheng Yu, Vincent Liu, and Qizhen Zhang. Cowbird: Freeing cpus to compute by offloading the disaggregation of memory. In Henning Schulzrinne, Vishal Misra, Eddie Kohler, and David A. Maltz, editors, *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, pages 1060–1073. ACM, 2023.

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[20] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)*, 17(4):1–32, 2021.

[21] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[22] Ulfar Erlingsson, Mark Manasse, and Frank McSherry. A cool and practical alternative to traditional hash tables. In *Proc. 7th Workshop on Distributed Data and Structures (WDAS'06)*, pages 1–6, 2006.

[23] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 249–264. USENIX Association, 2016.

[24] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, high-performance memory disaggregation with directcxl. In Jiri Schindler and Noa Zilberman, editors, *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 287–294. USENIX Association, 2022.

[25] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017*, pages 649–667, 2017.

[26] Jiasheng Hu, Philip Bernstein, Jialin Li, and Qizhen Zhang. DPDPU: Data Processing with DPUs. In *CIDR '25: Conference on Innovative Data Systems Research*. www.cidrdb.org, 2025.

[27] Jiasheng Hu, Chihan Cui, Anna Li, Raahil Vora, Yuanfan Chen, Philip A Bernstein, Jialin Li, and Qizhen Zhang. dpbento: Benchmarking dpus for data processing. *arXiv preprint arXiv:2504.05536*, 2025.

[28] Zhisheng Hu, Pengfei Zuo, Yizou Chen, Chao Wang, Junliang Hu, and Ming-Chang Yang. Aceso: Achieving efficient fault tolerance in memory-disaggregated key-value stores. In Emmett Witchel, Christopher J. Rossbach, Andrea C. Arpaci-Dusseau, and Kimberly Keeton, editors, *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024*, pages 127–143. ACM, 2024.

[29] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. Redplane: enabling fault-tolerant stateful in-switch applications. In Fernando A. Kuipers and Matthew C. Caesar, editors, *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 223–244. ACM, 2021.

[30] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostic, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, 2021*, pages 756–771.

[31] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. Farview: Disaggregated memory with operator off-loading for database engines. In *12th Conference on Innovative Data Systems Research, CIDR 2022*. www.cidrdb.org, 2022.

[32] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016*, pages 355–370.

[33] 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*, pages 267–278. ACM, 2009.

[34] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA , New Orleans, LA, USA, 25-29 February, 2012*, pages 189–200. IEEE Computer Society, 2012.

[35] Jiaxin Lin, Tao Ji, Xiangpeng Hao, Hokeun Cha, Yanfang Le, Xiangyao Yu, and Aditya Akella. Towards accelerating data intensive application's shuffle process using smartnics. In Evgenia Smirni, Konstantin Avrachenkov, Phillipa Gill, and Bhuvan Urgaonkar, editors, *Abstract Proceedings of the 2023 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2023, Orlando, FL, USA, June 19-23, 2023*, pages 9–10. ACM, 2023.

[36] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. Systematic CXL memory characterization and performance analysis at scale. In Lieven Eeckhout, Georgios Smaragdakis, Katai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach, editors, *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, pages 1203–1217. ACM, 2025.

[37] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. DEX: scalable range indexing on disaggregated memory. *Proc. VLDB Endow.*, 17(10):2603–2616, 2024.

[38] Xuchuan Luo, Jiacheng Shen, Pengfei Zuo, Xin Wang, Michael R. Lyu, and Yangfan Zhou. CHIME: A cache-efficient and high-performance hybrid index on disaggregated memory. In Emmett Witchel, Christopher J. Rossbach, Andrea C. Arpaci-Dusseau, and Kimberly Keeton, editors, *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*, pages 110–126. ACM, 2024.

[39] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. SMART: A high-performance adaptive radix tree for disaggregated memory. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 553–571. USENIX Association, 2023.

[40] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In Ada Gavrilovska and Erez Zadok, editors, *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 843–857. USENIX Association, 2020.

[41] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. Dremel: A decade of interactive SQL analysis at web scale. *Proc. VLDB Endow.*, 13(12):3461–3472, 2020.

[42] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In Fernando A. Kuipers and Matthew C. Caesar, editors, *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 106–122. ACM, 2021.

[43] 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.

[44] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 740–755, New York, NY, USA, 2021. Association for Computing Machinery.

[45] 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.

[46] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. FUSEE: A fully memory-disaggregated key-value store. In Ashvin Goel and Dalit Naor, editors, *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, pages 81–98. USENIX Association, 2023.

[47] Lasse Thostrup, Daniel Failing, Tobius Ziegler, and Carsten Binnig. A dbms-centric evaluation of bluefield dpus on fast networks. In Rajesh Bordawekar and Tirthankar Lahiri, editors, *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2022, Sydney, Australia, September 5, 2022*, pages 1–10, 2022.

[48] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. Dfi: the data flow interface for high-speed networks. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1825–1837, 2021.

[49] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In Zachary G. 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.

[50] Ruihong Wang, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. dlsm: An lsm-based index for memory disaggregation. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*, pages 2835–2849. IEEE, 2023.

[51] Chenyuan Wu, Mohammad Javad Amiri, Jared Asch, Heena Nagda, Qizhen Zhang, and Boon Thau Loo. Flexchain: An elastic disaggregated blockchain. *Proc. VLDB Endow.*, 16(1):23–36, 2022.

[52] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In Shan Lu and Erik Riedel, editors, *Proceedings of the 2015 USENIX Annual Technical Conference, USENIX ATC 2015, July 8-10, Santa Clara, CA, USA*, pages 71–82. USENIX Association, 2015.

[53] Ming Zhang, Yu Hua, and Zhijun Yang. Motor: Enabling multi-versioning for distributed transactions on disaggregated memory. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 801–819. USENIX Association, 2024.

[54] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. Redy: Remote dynamic memory cache. *Proc. VLDB Endow.*, 15(4):766–779, 2022.

[55] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. Compucache: Remote computable caching using spot vms. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022.

[56] Qizhen Zhang, Philip A. Bernstein, Badrish Chandramouli, Jason Hu, and Yiming Zheng. DDS: dpu-optimized disaggregated storage. *Proc. VLDB Endow.*, 17(11):3304–3317, 2024.

[57] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. Rethinking data management systems for disaggregated data centers. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[58] 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.

[59] 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 G. 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.

[60] 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.

[61] Yuhong Zhong, Daniel S. Berger, Carl A. Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with CXL in virtualized environments. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 37–56. USENIX Association, 2024.

[62] Wenbin Zhu, Zhaoyan Shen, Qian Wei, Renhai Chen, Xin Yao, Dongxiao Yu, and Zili Shao. Hidpu: A dpu-oriented hybrid indexing scheme for disaggregated storage systems. In Haryadi S. Gunawi and Vasily Tarasov, editors, *23rd USENIX Conference on File and Storage Technologies, FAST 2025, Santa Clara, CA, February 25-27, 2025*, pages 271–285. USENIX Association, 2025.

[63] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 741–758. ACM, 2019.

[64] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided rdma-conscious extendible hashing for disaggregated memory. In Irina Calciu and Geoff Kuenning, editors, *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 15–29. USENIX Association, 2021.

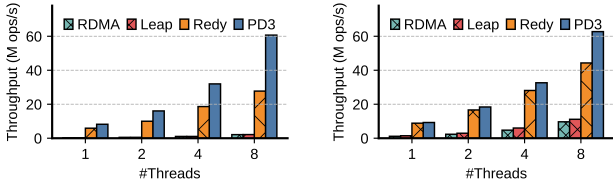


Figure B.2: **Throughput of hash map (YCSB, uniform).**

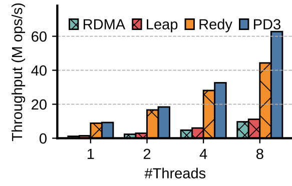


Figure B.3: **Throughput of hash map (YCSB, Zipfian).**

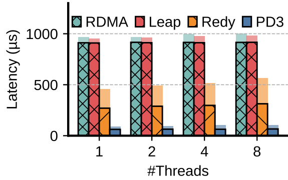


Figure B.4: **Latency of hash map (YCSB, uniform).**

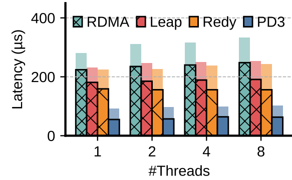
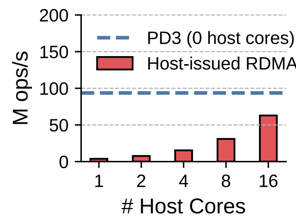
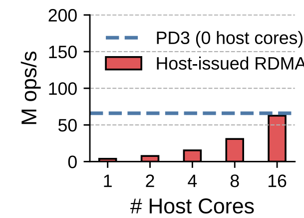


Figure B.5: **Latency of hash map (YCSB, Zipfian).**



(a) Transferring data to the DPU



(b) Transferring data to memory

Figure A.1: **Data transfer speed and host CPU consumption comparison between PD3 and host-based solution.**

A Performance of Transfer Engine

The results of transferring 64 B records from the host to the DPU and from the host to the remote memory server comparing PD3's Transfer Engine and host-issued RDMA are shown in Figure A.1. Specifically, Transfer Engine achieves more with less: 94 million ops/s throughput without host resource consumption, while host-issued RDMA achieves 63 million ops/s and consumes 16 CPU cores (Figure A.1a). In terms of remote memory access performance, compared to host-issued RDMA, Transfer Engine consumes no host cores but achieves the same throughput as host-issued RDMA consuming 16 host CPU cores.

B Additional YCSB Results

This section shows additional experimental results for the hashmap application with the YCSB-C workload (read-only).

Figures B.2 and B.3 show the throughput for PD3 and the compared approaches of synchronous RDMA, Leap and Redy. Synchronous RDMA shows the lowest performance on both workloads. Leap improves throughput on the skewed workload as the prefetcher effectively predicts accesses. In the face of unpredictable accesses, Leap does not prefetch at all to prevent cache pollution, and therefore does not improve performance on the uniform workload. Redy provides highly optimized remote memory access, increasing performance by up to an order of magnitude, but still suffers from the overhead of the remote memory operations. PD3 sidesteps this overhead with the early access prefetching path and therefore achieves up to 2.3× higher throughput than Redy on the uniform workload.

Figures B.4 and B.5 show the latency for the hashmap application under the same workloads. Since PD3's early access prefetching path results in memory accesses being local instead of remote, it greatly reduces latency compared to other approaches. Compared to synchronous RDMA and Leap, PD3 reduces latency by up to 11× and compared to Redy, PD3 reduces latency by up to 5×.

The results on this workload are similar to the ones for YCSB Workload B (reported in Section 8.2) showing PD3's ability to maintain high performance in both read-only and mixed workloads.