PULSE: Accelerating Distributed Pointer-Traversals on Disaggregated Memory

Yupeng Tang
Yale University
New Haven, United States

Seung-seob Lee
Yale University
New Haven, United States

Abhishek Bhattacharjee
Yale University
New Haven, United States

Anurag Khandelwal
Yale University
New Haven, United States

Abstract

Caches at CPU nodes in disaggregated memory architectures amortize the high data access latency over the network. However, such caches are fundamentally unable to improve performance for workloads requiring pointer traversals across linked data structures. We argue for accelerating these pointer traversals closer to disaggregated memory in a manner that preserves expressiveness for supporting various linked structures, ensures energy efficiency and performance, and supports distributed execution. We design PULSE, a distributed pointer-traversal framework for rack-scale disaggregated memory to meet all the above requirements. Our evaluation of PULSE shows that it enables low-latency, high-throughput, and energy-efficient execution for a wide range of pointer traversal workloads on disaggregated memory that fare poorly with caching alone.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Hardware** → **Networking hardware**; • **Information systems** → **Data structures**.

Keywords: Disaggregated memory, Pointer-traversals, Near-memory processing, Programmable networks, FPGAs

ACM Reference Format:
Yupeng Tang, Seung-seob Lee, Abhishek Bhattacharjee, and Anurag Khandelwal. 2025. PULSE: Accelerating Distributed Pointer-Traversals on Disaggregated Memory. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3669940.3707253

1 Introduction

Driven by increasing demands for memory capacity and bandwidth [35, 50, 54, 112, 133, 154, 160], poor scaling [85, 99, 134] and resource inefficiency [72, 122] of DRAM, and improvements in Ethernet-based network speeds [12, 39], recent years have seen significant efforts towards memory disaggregation [42, 46, 72, 100, 130]. Rather than scaling up a server's DRAM capacity and bandwidth, such proposals advocate disaggregating much of the memory over the network. The result is a set of CPU nodes equipped with a small amount of DRAM used as cache[1], accessing memory across a set of network-attached memory nodes with large DRAM pools (Fig. 1 (top)). With allocation flexibility across CPU and memory nodes, disaggregation enables high utilization and elasticity. Despite drastic improvements in recent years, the limited bandwidth and latency to network-attached memory remain a hurdle in adopting disaggregated memory, with speed-of-light constraints making it impossible to improve network latency beyond a point. Even with near-terabit links and hardware-assisted protocols like RDMA [10], remote memory accesses are an order of magnitude slower than local memory accesses [65]. Emerging CXL interconnects [24] share a similar trend — around 300 ns of CXL memory latency compared to 10–20 ns of L3 cache latency [101]. Although efficient caching strategies at the CPU node can reduce average memory access latency and volume of network traffic to remote memory, the benefit of such strategies is limited by data locality and the size of the cache on the CPU node. In many cases, remote memory accesses are unavoidable, especially for applications that rely on efficient in-memory pointer traversals on linked data structures, such as lookups on index structures [31, 32, 38, 48, 49, 53, 75, 89, 106, 109, 161] in databases and key-value stores, and traversals in graph analytics [67, 68, 97, 118] (Fig. 2, §2).

Similar to how CPUs have small but fast memory (i.e., caches) for quick access to popular data, we argue that memory nodes should also include lightweight but fast processing units with high-bandwidth, low-latency access to memory to speed up pointer-traversals (Fig. 1 (bottom)). Moreover, the interconnect should facilitate efficient and scalable distributed traversals for deployments with multiple memory nodes that cater to large-scale linked data structures. Prior works have explored systems and API designs for such processing units under multiple settings, ranging

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707253

[1]Not to be confused with die-stacked hardware DRAM caches [80, 81, 157].

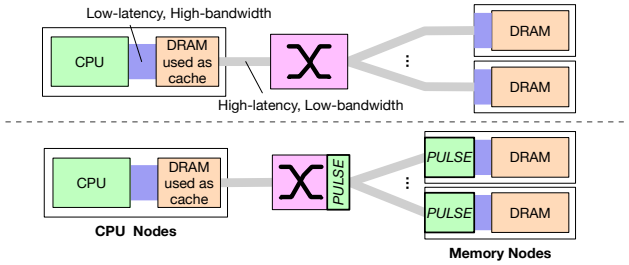


Fig. 1. Need for accelerating pointer traversals. *(top)* The performance of pointer traversals in disaggregated architectures is bottlenecked by slow memory interconnect. *(bottom)* Just as caches offer limited but fast memory near CPUs, we argue that memory needs a counterpart for traversal-heavy workloads: a lightweight but fast accelerator for cache-unfriendly pointer traversals.

from near-memory processing and processing-in-memory approaches [40, 47, 52, 56, 57, 59–62, 66, 73, 76, 86, 87, 90, 96, 105, 110, 111, 115, 116, 128, 129, 136, 142, 148, 149, 151, 152] for single-server architectures, to the use of CPUs [51, 94, 113, 127, 156, 162] or FPGAs [74, 135] near remote/disaggregated memory, but have several key shortcomings.

Specifically, existing approaches are limited in scale and expose a three-way tradeoff between expressiveness, energy efficiency, and performance. First, and perhaps most crucially, none of the existing approaches can accelerate pointer traversals that span *multiple* network-attached memory nodes.

This limits memory utilization and elasticity since applications must confine their data to a single memory node to accelerate pointer traversals. Their inability to support distributed pointer traversals stems from complex management of address translation state that is required to identify if a traversal can occur locally or must be re-routed to a different memory node (§2.2). Second, existing single-node approaches use full-fledged CPUs for expressive and performant execution of pointer-traversals [51, 94, 127, 156]. However, coupling large amounts of processing capacity with memory — which has utility in reducing data movement in PIM architectures [40, 47, 59, 61, 62, 105, 110, 111, 116, 128, 142, 148, 151] — goes against the very spirit of memory disaggregation since it leads to poor utilization of compute resources and, consequently, poor energy efficiency.

Approaches that use wimpy processors at SmartNICs [43, 120] instead of CPUs retain expressiveness, but the limited processing speeds of wimpy nodes curtail their performance and, ultimately lead to lower energy efficiency due to their lengthened executions (§6.1, [74]). Lastly, FPGA-based [74, 135, 138] and ASIC-based [76, 90] approaches achieve performance and energy efficiency by hard-wiring pointer traversal logic for specific data structures, limiting their expressiveness.

We design PULSE[2], a distributed pointer-traversal framework for rack-scale disaggregated memory, to meet all of the above needs — namely, expressiveness, energy efficiency, performance — via a principled redesign of near-memory processing for disaggregated memory. Central to PULSE's design is an expressive iterator interface that readily lends itself to a unifying abstraction across most pointer traversals in linked data structures used in key-value stores [30, 121], databases [49, 53, 75, 108, 109], and big-data analytics [67, 68, 97, 118] (§3). PULSE's use of this abstraction not only makes it immediately useful in this large family of real-world traversal-heavy use cases, but also enables (i) the use of familiar compiler toolchains to support these use cases with little to no application modifications and (ii) the design of tractable hardware accelerators and efficient distributed traversal mechanisms that exploit properties unique to iterator abstractions.

In particular, PULSE enables transparent and efficient execution of pointer traversals for our iterator abstraction via a novel accelerator that employs a *disaggregated* architecture to decouple logic and memory pipelines, exploiting the inherently sequential nature of compute and memory accesses in iterator execution (§4). This permits high utilization by provisioning more memory and fewer logic pipelines to cater to memory-centric pointer traversal workloads. A scheduler breaks pointer traversal logic from multiple concurrent workloads across the two sets of pipelines and employs a novel multiplexing strategy to maximize their utilization. While our implementation leverages an FPGA-based SmartNIC due to the high cost and complexity of ASIC fabrication, our ultimate vision is an ASIC-based realization for improved performance and energy efficiency.

We enable distributed traversals by leveraging the insight that pointer traversal across network-attached memory nodes is equivalent to packet routing at the network switch (§5). As such, PULSE leverages a programmable network switch to inspect the next pointer to be traversed within iterator requests and determine the next memory node to which the request should be forwarded — both at line rate.

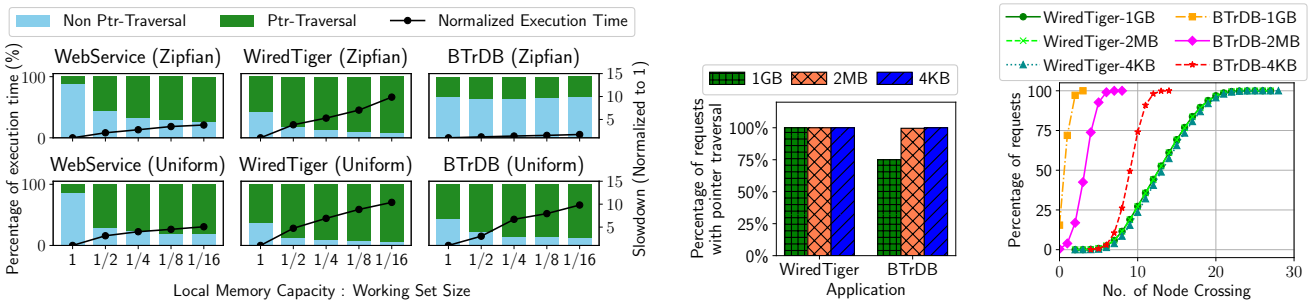
We implement a real-system prototype of PULSE on a disaggregated rack of commodity servers, SmartNICs, and a programmable switch with full-system effects. None of PULSE's hardware or software changes are invasive or overly complex, ensuring deployability. Our evaluation of end-to-end real-world workloads shows that PULSE outperforms disaggregated caching systems with 9–34× lower latency and 28–171× higher throughput. Moreover, our Xilinx XRT [37] and Intel RAPL [78]-based power analysis shows that PULSE consumes 4.5–5× less energy than RPC-based schemes (§6).

2 Motivation and PULSE Overview

2.1 Need for Accelerating Pointer Traversals

Memory-intensive applications [35, 50, 54, 112, 133, 154, 160] often require traversing linked structures like lists, hash tables, trees, and graphs. While disaggregated architectures

[2]**P**rocessing **U**nit for **L**inked **S**tructur**E**s.



(a) Our empirical analysis (b) % of distributed traversals (c) CDF of distributed traversals

Fig. 2. Time cloud applications spend in pointer traversals. See §2.1 for details.

provide large memory pools across network-attached memory nodes, traversing pointers over the network is still slow [65]. Recent proposals [42, 65, 72, 100, 130] alleviate this slowdown by using the DRAM at the CPU nodes to cache "hot" data, but such caches often fare poorly for pointer traversals, as we show next.

Pointer traversals in real-world workloads. Prior studies [30, 34, 63, 77, 97, 158, 160] have shown that real-world data-centric cloud applications spend anywhere from 21% to 97% of execution time traversing pointers. We empirically analyze the time spent in pointer traversals for three representative cloud applications — a WebService frontend [127], indexing on WiredTiger [108], and time-series analysis on BTrDB [45] — with swap-based disaggregated memory [72][3]. We vary the cache size at the CPU node from 6.25%-100% of each application's working set size. Fig. 2(a) shows that (i) all three applications spend a significant fraction of their execution time (13.6%, 63.7%, and 55.8%, respectively) traversing pointers even when their entire working set is cached, and (ii) the time spent traversing pointers (and thus, the end-to-end execution time) increases with smaller CPU node caches. While the impact of access skew is application-dependent, pointer traversals dominate application execution times when more of the application's working set size is remote.

Distributed traversals. As the number of applications and the working-set size per application grows larger, disaggregated architectures must allocate memory across multiple memory nodes to keep up. Such approaches [42, 72, 100, 130] tend to strive for the smallest viable allocation granularity with reasonable metadata overheads (e.g., 1 GB in [130], 2 MB in [100]) since smaller allocations permit better load balancing and high memory utilization. Unfortunately, finer-grained allocations may cause an application's linked structures to get fragmented across multiple network-attached memory nodes, necessitating many *distributed* traversals.

[3]We defer the details of the data structures and workloads employed by these applications, as well as the disaggregated memory setup to §6.

Fig. 2(b) illustrates this impact on a setup with 1 compute and 4 memory nodes: even with large 1 GB allocations, WiredTiger and BTrDB require over 97% and 75% of their requests, respectively, to cross memory node boundaries at least once, with the volume of cross-node traffic increasing at smaller granularities. Fig. 2(c) shows the CDF of requests that require a certain number of memory node crossings. While the randomly ordered data in WiredTiger necessitate many cross-node traversals even for large allocations, the time-ordered data in BTrDB reduce cross-node traversals for larger allocation granularities by confining large time windows to the same memory node. However, smaller to moderate allocation granularities — required for high memory utilization — still require many cross-node traversals.

2.2 Shortcomings of Prior Approaches

No prior work achieves all four properties required for pointer traversals on disaggregated memory: distributed execution, expressiveness, energy efficiency, and performance. We focus on network-attached memory, although a similar analysis extends to in-memory processing [40, 47, 52, 56, 57, 59–62, 66, 73, 76, 86, 87, 90, 96, 105, 110, 111, 115, 116, 128, 129, 136, 142, 148, 149, 151, 152].

No support for distributed execution. Distributed pointer traversals are required to ensure applications can efficiently access large pools of network-attached memory nodes. Unfortunately, to our knowledge, none of the prior works support efficient multi-node pointer traversals. Therefore, applications must confine their data to a single node for efficient traversals, exposing a tradeoff between application performance and scalability. Recent proposals [44, 104, 107, 131, 132, 141, 146] explore specialized data structures that co-design partitioning and allocation policies to reduce distributed pointer traversals atop disaggregated memory. Such approaches complement our work since they still require efficient distributed traversals when their optimizations are not applicable, *e.g.*, not many data structures benefit from such specialized co-designs.

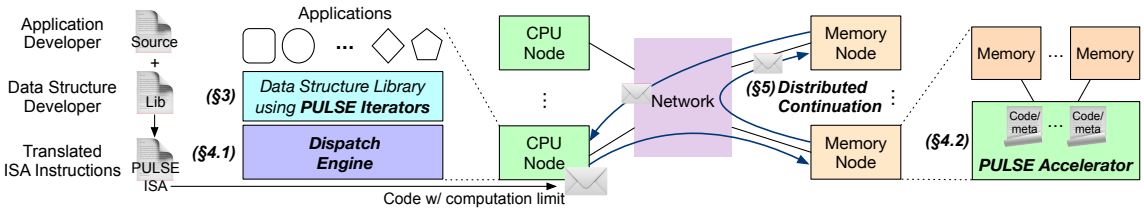


Fig. 3. PULSE Overview. Developers use PULSE's iterator interface (§3) to express pointer traversals, translated to PULSE ISA by its dispatch engine (§4.1). During execution, PULSE accelerator ensures energy efficiency (§4.2) and in-network design enable distributed traversals (§5).

Poor performance with prefetching approaches. Cache-based designs for remote memory often employ prefetching techniques [41, 103, 155] that pipeline remote memory access with computations at the CPU nodes. Unfortunately, such pipelining does not improve performance for pointer traversal workloads for two main reasons. First, the remote memory access latency is typically far greater than the computation required for pointer traversals, so network round trips to prefetch the data would remain the bottleneck. Second, speculating the next unit of data to prefetch for more complex data structures like B+Trees or graphs, where each node contains pointers to many "children" nodes, tends to have much lower accuracy in practice. As such, prefetching can even add overheads due to unnecessary data fetches.

Poor utilization/power-efficiency in CPUs. Many prior works have explored remote procedure calls (RPCs) to enable offloading computation to CPUs on memory nodes [51, 94, 113, 127, 156]. While CPUs are performant and versatile enough to support most general-purpose computations, the same versatility makes them overkill for pointer traversal workloads in disaggregated architectures — the CPUs on memory nodes are likely to be underutilized and, consequently, waste energy (§6), since such workloads are memory-intensive and bounded by memory bandwidth rather than CPU cycles.

Since inefficient power usage resulting from coupled compute and memory resources is the main problem disaggregation aims to resolve, leveraging CPUs at memory nodes essentially nullifies these benefits.

Limited expressiveness in FPGA/ASIC accelerators. Another approach explored in recent years uses FPGAs [74, 135] or ASICs [76, 90] at memory nodes for performance and energy efficiency. FPGA approaches exploit circuit programmability to realize performant on-path data processing, albeit only for specific data structures, limiting their expressiveness. Although some FPGA approaches aim for greater expressiveness by serving RPCs [92], RPC logic must be pre-compiled before it is deployed and physically consumes FPGA resources. This limits how many RPCs can be deployed on the FPGA concurrently and also elides runtime resource elasticity for different pointer traversal workloads. ASIC approaches either support a single data structure or provide limited ISA specialized for a single data structure (*e.g.*, linked-lists [90]), limiting their general applicability.

Poor performance/power efficiency in wimpy SmartNICs. The emergence of programmable SmartNICs has driven work on offloading computations to the onboard network processors. Some approaches utilize wimpy processors (*e.g.*, ARM or RISC-V processors) [43] or RDMA processing units (PUs) [120] to support general-purpose computations near memory. While these wimpy processors can eliminate multiple network round trips in pointer traversal workloads, their processing speeds are far slower than CPU-based or FPGA-based accelerators. Often, such PUs can become a performance bottleneck, especially at high memory bandwidth (~500 Gbps) [65, 120]. Moreover, wimpy processors tend not to be energy-efficient since their slower execution tends to waste more static power, resulting in higher energy per pointer traversal offload — an observation noted in prior work [74] and confirmed in our evaluation (§6).

2.3 PULSE Design Overview

PULSE innovates on three key design elements (Fig. 3). Central to PULSE's design is its iterator-based programming model (§3) that requires minimal effort to port real-world data structure traversals. PULSE supports *stateful* traversals using a *scratchpad* of pre-configured size, where developers can store and update arbitrary intermediate states (*e.g.*, aggregators, arrays, lists, etc.) during the iterator's execution. Properties specific to iterator patterns enable tractable accelerator design and efficient distributed traversals in PULSE.

The iterator code provided by the data structure developer is translated into PULSE's instruction set architecture (ISA) to be executed by PULSE accelerators (§4). PULSE achieves energy efficiency and performance through a novel accelerator that employs disaggregated logic and memory pipelines and an ISA specifically designed for the iterator pattern. Our accelerator employs a scheduler specialized for its disaggregated architecture to ensure high utilization *and* performance.

PULSE supports scalable distributed pointer traversals by leveraging programmable network switches to reroute any requests that must cross memory node boundaries (§5). PULSE employs hierarchical address translation *in the network*, where memory node-level address translation is performed at the switch (i.e., a request is routed to the memory

node based on its target address), and the memory node accelerator performs translation and protection for local accesses. During traversal, a memory node accelerator can return a request to the switch if it determines the address is not local; the switch re-routes the request to the correct memory node.

Assumptions. PULSE does not offload synchronization to its accelerators but instead requires the application logic at the CPU node to explicitly acquire/release appropriate locks for the offloaded operation. Recent efforts enable locking primitives on NICs [141, 146] and programmable switches [159]; these are orthogonal to our work and can be incorporated into PULSE. Finally, PULSE does not innovate on caching and adapts the caching scheme from prior work [127], which maintains a transparent cache within the data structure library.

3 PULSE Programming Model

We begin with PULSE's programming model since a carefully crafted interface is crucial to enable wide applicability for real-world traversal-heavy applications, as well as the design of tractable pointer traversal accelerators and efficient distributed traversal mechanisms. PULSE's interface is intended for data structure library developers to offload pointer traversals in linked data structures. Since PULSE code modifications are restricted to data structure libraries, existing applications utilizing their interfaces require no modifications.

We analyzed the implementations of a wide range of popular data structures [2, 15, 16, 28] to determine the structures common to them in pointer traversals. We found that most traversals (1) initialize a start pointer using data structure-specific logic, (2) iteratively use data structure-specific logic to determine the next pointer to look up, and (3) check a data structure-specific termination condition at the end of each iteration to determine if the traversal should end. This structure resembles that of the *iterator* design pattern, establishing its universality as a design motif common to almost all languages [28]. This is precisely what makes it an ideal candidate for the interface between the hardware and software layers for pointer traversals. As such, PULSE allows developers to program their data structure traversals using the iterator interface shown in Listing 1.

The interface exposes three functions that must be implemented by the user: (1) `init()`, which takes as input arbitrary data structure-specific state to initialize the start pointer, (2) `next()`, that updates the current pointer to the next pointer it must traverse to, and, (3) `end()`, that determines if the pointer traversal should end (either in success or failure) based on the current pointer. PULSE then uses the provided implementations for these functions to execute the pointer traversal iteratively, using the `execute()` function. We discuss two key novel aspects of our iterator abstraction that were necessary to increase and limit the expressiveness of operations on linked data structures.

```
1  class pulse_iterator {
2      void init(void *) = 0; // Implemented by developer
3      void *next() = 0; // Implemented by developer
4      bool end() = 0; // Implemented by developer
5
6      unsigned char *execute() { // Non-modifiable logic
7        unsigned int num_iter = 0;
8        while (!end() && num_iter++ < MAX_ITER)
9          cur_ptr = next();
10       return scratch_pad;
11     }
12     uintptr_t cur_ptr;
13     unsigned char scratch_pad[MAX_SCRATCHPAD_SIZE];
14 }
```

Listing 1. PULSE interface.

Stateful traversals. Pointer traversals in many data structures are stateful, and the nature of the state can vary widely. For instance, in hash table lookups, the state is the search key that must be compared against a linked list of keys in a hash bucket. In contrast, summing up values across a range of keys in a B-Tree requires maintaining a running variable for storing the sum and updating it for each value encountered in the range. To facilitate this, PULSE iterators maintain a `scratch_pad` that the developer can use to store an arbitrary state. The `scratch_pad` acts as a continuation [123] in the programming language sense, allowing the state to persist across iterations. It is initialized in `init()`, updated in `next()`, and finalized in `end()`. Since `execute()` in PULSE's iterator interface returns the contents of `scratch_pad` (Line 10), developers can place the state they want to retrieve in it.

Bounded computations. PULSE accelerators support only lightweight processing in memory-intensive operations for high memory bandwidth utilization. While `init()` is executed on the CPU node, `next()` and `end()` are offloaded to PULSE accelerators; hence, PULSE limits what memory accesses and computations can be performed in them in two ways. Within each iteration, PULSE disallows nondeterministic executions, such as unbounded loops, i.e., loops that cannot be unrolled to a fixed number of instructions. Across iterations, `execute()` in Listing 1 limits the maximum number of iterations that a single request is allowed to perform. This ensures that a particularly long traversal does not block other requests for a long time.

If a request exceeds the maximum iteration count, PULSE terminates the traversal and returns the `scratch_pad` value to the CPU node, which can issue a new request to continue the traversal from that point.

An illustrative example. We demonstrate how the `find()` operation on C++ STL `unordered_map` can be ported to PULSE. Listing 2 shows a simplified version of its implementation in STL — the pointer traversal begins by computing a hash function and determining a pointer to the hash bucket corresponding to the hash. It then iterates through a linked list

```
struct node {
  key_type key;
  value_type value;
  struct node *next;
};

value_type find(key_type key) {
  for (struct node *cur_ptr = bucket_ptr(hash(key));
       ; cur_ptr = cur_ptr->next) {
    if (key == cur_ptr->key) // Key found
      return cur_ptr->value;
    if (cur_ptr->next == nullptr) // Key not found
      break;
  }
  return KEY_NOT_FOUND;
}
```

Listing 2. C++ STL realization for `unordered_map::find()`.

```
class unordered_map_find : pulse_iterator {
  init(void *key) {
    memcpy(scratch_pad, key, sizeof(key_type));
    cur_ptr = bucket_ptr(hash((key_type)*key));
  }

  void* next() { return cur_ptr->next; }

  bool end() {
    key_type key = *((key_type *)scratch_pad);
    if (key == cur_ptr->key) { // Key found
      *((value_type *)scratch_pad) = cur_ptr->value;
      return true;
    }
    if (cur_ptr->next == nullptr) { // Key not found
      *((unsigned int *)scratch_pad) = KEY_NOT_FOUND;
      return true;
    }
    return false;
  }
}
```

Listing 3. PULSE realization for `unordered_map::find()`.

corresponding to the hash bucket, terminating if the key is found or the linked list ends without it being found.

Listing 3 shows the corresponding iterator implementation in PULSE. Much of the implementation is unchanged, with minor restructuring for `init()`, `next()`, and `end()` functions. The main changes are — how the state (the search key) is exchanged across the three functions and how the data is returned back to the user via the `scratch_pad` (an error message if the key is not found, or its value if it is).

Ported Data Structures. While the PULSE programming model applies to various programming languages that support iterator interfaces, we have restricted our focus to C++ data structure libraries due to their widespread use. We have applied the PULSE programming model to 13 commonly-used data structures found in popular libraries such as STL [16], Boost [2], and Google Btree [7] (Table 1); we defer a comprehensive description of their details to [139].

4 Accelerating Pointer Traversals on a Node

4.1 PULSE Dispatch Engine

The dispatch engine is a software framework running at the CPU node for two purposes. First, it translates the iterator realization for pointer traversal provided by a data structure library developer (§3) into PULSE's ISA. Second, it determines if the accelerator can support the computations performed during the traversal, and if so, ships a request to the accelerator at the memory node. If not, the execution proceeds at the CPU node with regular remote memory accesses.

Translating iterator code to PULSE ISA. To be readily implementable, PULSE plugs into existing compiler toolchains. The dispatch engine generates PULSE ISA instructions using widely known compiler techniques [33]. PULSE's ISA is a stripped-down RISC ISA, only containing operations necessary for basic processing and memory accesses to enable a simple and energy-efficient accelerator design (Table 2). There are, however, a few notable aspects to our adapted ISA and the translation of iterator code to it. First, as noted in §3, PULSE does not support unbounded loops within a single iteration, i.e., the ISA only supports conditional jumps to points ahead in code. This is similar to eBPF programs [124], where only forward jumps are supported to prevent the program from running infinitely within the kernel. A backward jump can only occur when the next iteration starts; PULSE employs a special `NEXT_ITER` instruction to explicitly mark this point so that the accelerator can begin scheduling the memory pipeline (§4.2). Second, again as noted in §3, developers can maintain state and return values using a `scratch_pad` of preconfigured size; our ISA supports register operations directly on the `scratch_pad` and provides special `RETURN` instruction that simply terminates the iterator execution and yields the contents of the `scratch_pad` as the return value. Lastly, if the code cannot be compiled to the PULSE ISA — *e.g.*, if it involves compute-heavy or non-memory-centric tasks — it will not be offloaded to the accelerator. Instead, such code will run on the CPU, potentially accessing memory remotely over the network (*e.g.*, via RDMA or CXL). This design ensures that only tasks that benefit from near-memory execution are offloaded, adhering to our design philosophy of only offloading memory-bound operations.

Finally, we found that the iterator traversal pattern typically can be broken down into two types of computation — fetching data[4] pointed to by `cur_ptr` from memory, and processing the fetched data to determine what the next pointer should be, or if the iterator execution should terminate. If the translation from the iterator code to PULSE's ISA is done naively, it can result in multiple unnecessary loads within the vicinity of the memory location pointed to

[4]While the rest of the section focuses only on describing data fetches from memory, we note that writing data to memory proceeds similarly.

Library	Data Structures
STL	List [19], Forward list [18], Map [20], Multimap [21], Set [23], Multiset [22]
Boost	Bimap [14], Unordered map [5], Unordered set [6] AVL tree [1], Splay tree [3], Scapegoat tree [4],
Google	Btree [7]

Table 1. Data structures implemented in PULSE (§3).

Class	Instructions	Description
Memory	`LOAD`, `STORE`	Load/store data from/to address.
ALU	`ADD`, `SUB`, `MUL`, `DIV`, `AND`, `OR`, `NOT`	Standard ALU operations.
Register	`MOVE`	Move data b/w registers.
Branch	`COMPARE` and `JUMP_{EQ, NEQ, LT, ...}`	Compare values & jump ahead based on condition (*e.g.*, equal, less than, etc.).
Terminal	`RETURN`, `NEXT_ITER`	End traversal & return, or start next iteration.

Table 2. PULSE adapts a restricted subset of RISC-V ISA (§4.1).

by `cur_ptr`. For instance, the `unordered_map::find()` realization shown in Listing 3 makes references to `cur_ptr->key`, `cur_ptr->value`, and `cur_ptr->next` at various points, and if each incurs a separate load, it will slow down execution and waste memory bandwidth. Consequently, PULSE's dispatch engine *infers* the range of memory locations accessed relative to `cur_ptr` in the `next()` and `end()` functions via static analysis and aggregates these accesses into a single large `LOAD` (of up to 256 B) at the beginning of each iteration.

Bounding complexity of offloaded code. While PULSE's interface and ISA already limit the *types* of computation than can be performed per iteration, PULSE also needs to limit the *amount* of computation per iteration to ensure the operations offloaded to PULSE accelerators remain memory-centric. To this end, PULSE's dispatch engine analyzes the generated ISA for the iterator to determine the time required to execute computational logic (t_c) and the time required to perform the single data load at the beginning of the iteration (t_d). PULSE exploits the known execution time of its accelerators in terms of time per compute instruction, t_i, to determine $t_c = t_i \cdot N$, where N is the number of instructions per iteration. The CPU node offloads the iterator execution only if $t_c \leq \eta \cdot t_d$, where η is a predefined accelerator-specific threshold. Note that since we only want to offload memory-centric operations, $\eta \leq 1$. As we will show in §4.2, the choice of η allows PULSE to maximize the memory bandwidth utilization and ensure processing never becomes a bottleneck for pointer traversals.

Issuing network requests to accelerator. Once the dispatch engine decides to offload an iterator execution, it encapsulates the ISA instructions (`code`) along with the initial value of `cur_ptr` and `scratch_pad` (initialized by `init()`) into a network request. It issues the request, leaving the network to determine which memory node it should be forwarded to (§5). To recover from packet drops, the dispatch engine embeds a request ID with the CPU node ID and a local request counter in the request packets, maintains a timer per request, and transparently retransmits requests on timeout.

Practical deployability. Our software stack is readily deployable due to its use of real-world toolchains. Our user library adapts implementations of common data structures used in key-value stores [30, 121], databases [49, 53, 75, 108, 109], and big-data analytics [67, 68, 97, 118] to PULSE's iterator interface (§3). PULSE's dispatch engine is implemented on Intel DPDK-based [26] low-latency, high-throughput UDP stack. PULSE compiler adapts the Sparc backend of LLVM [98] since its ISA is close to PULSE's ISA. Our LLVM frontend applies a set of analysis and optimization passes [29] to enforce PULSE constraints and semantics: the analysis pass identifies code snippets that require offloading, while the optimization pass translates pointer traversal code to PULSE ISA.

4.2 PULSE Accelerator Design

The accelerator is at the heart of PULSE design and is key to ensuring high performance for iterator executions with high resource and energy efficiency. Our motivation for a new accelerator design stems from two unique properties of iterator executions on linked structures:

- **Property 1:** Each iteration involves two clearly separated but sequentially dependent steps: (i) fetching data from memory via a pointer (*e.g.*, a list or tree node), followed by (ii) executing logic on the fetched data to identify the next pointer. The logic cannot be executed concurrently with or before the data fetch, and the next data fetch cannot be performed until the logic execution yields the next pointer.
- **Property 2:** Iterators that benefit from offload spend more time in data fetch (t_d) than logic execution (t_c), i.e., $t_c < \eta \cdot t_d$, where $\eta \leq 1$, as noted in §4.1.

Any accelerator for iterator executions must have a *memory pipeline* and a *logic pipeline* to support the execution steps (i) and (ii) above. The strict dependency between the steps (Property 1) renders many optimizations of traditional multi-core processors, such as out-of-order execution, ineffective. Moreover, since each core in such architectures has tightly coupled logic and memory pipelines, the memory-intensive nature of iterators (Property 2) results in the logic pipeline remaining idle most of the time. These two factors combined result in poor utilization and energy efficiency for such architectures. Fig. 4 (top) captures this through the execution of 3 iterators (A, B, C), each with 2 iterations (*e.g.*, A1, A2, etc.), on a multi-core architecture. Since each iteration comprises a data fetch followed by a dependent logic execution, one of the pipelines remains idle while the other is busy. While thread-level parallelism permits iterator requests to

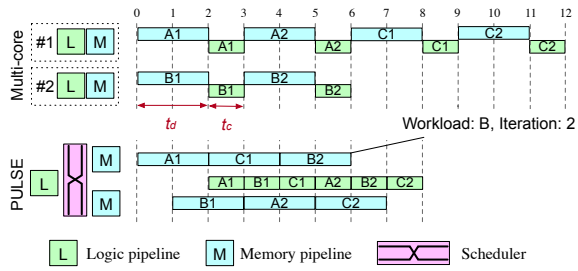


Fig. 4. PULSE accelerator architecture. (top) Traditional multi-core architectures with tightly coupled logic and memory pipelines result in low utilization and longer execution times. (bottom) PULSE accelerator's *disaggregated* design with an unequal number of logic and memory pipelines efficiently multiplexes concurrent iterator executions across them for near-optimal utilization and performance.

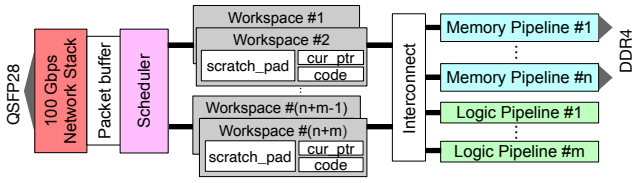


Fig. 5. PULSE accelerator overview. See §4.2 for details.

be spread across multiple cores for increased overall throughput, per-core under-utilization of logic and memory pipelines persists, resulting in suboptimal resource and energy usage.

Disaggregated accelerator design. Motivated by the unique properties of iterators, we propose a novel accelerator architecture that *disaggregates memory and logic pipelines*, using a scheduler to multiplex corresponding components of iterators across them. First, such a decoupling permits an asymmetric number of logic and memory pipelines to maximize the utilization of either pipeline, in stark contrast to the tight coupling in multi-core architectures. In our design, if there are m logic and n memory pipelines, then the accelerator-specific threshold $\eta < 1$ we alluded to in §4.1 is $\frac{m}{n}$, i.e., there are fewer logic pipelines than memory pipelines in keeping with Property 2. Fig. 4 (bottom) shows an example of our disaggregated accelerator design with one logic pipeline and two memory pipelines (i.e., $m = 1, n = 2$).

Even though data fetch and logic execution within each iterator must be sequential, the disaggregated design permits efficient multiplexing of data fetch and logic execution from different iterators across the disaggregated logic and memory pipelines to maximize utilization. To see how, recall that the logic execution time t_c for each offloaded iterator execution in PULSE is $\leq \eta \cdot t_d$, where t_d is its data fetch time (§4.1). Consider the extreme case where $t_c = \eta \cdot t_d$ for all offloaded iterator executions — in this case, it is always possible to multiplex $m + n$ concurrent iterator executions to fully utilize all m logic and n memory pipelines. While we omit a theoretical proof for brevity, Fig. 4 (bottom) illustrates the multiplexed execution — orchestrated by a scheduler in our accelerator — for $t_c = \frac{1}{2} \cdot t_d$ with 3 iterators. This is the ideal case — similar multiplexing is still possible if $t_c \leq \eta \cdot t_d$ with complete utilization of memory pipelines, albeit with lower utilization of logic pipelines (since they will be idle for $\frac{t_c - \eta \cdot t_d}{t_c}$ fraction of time). As such, we provision $\eta = \frac{m}{n}$ to be as close to the expected $\frac{t_c}{t_d}$ for the workload to maximize the utilization of logic pipelines. It is possible to improve the logic pipelines' energy efficiency by dynamically downscaling frequency [93]; we leave such optimizations to future work.

While the memory pipeline is stateless, the logic pipeline must maintain the state for the iterator it executes. To multiplex several iterator executions, logic pipelines need efficient mechanisms for efficient context switching. To this end, we maintain a dedicated *workspace* corresponding to each iterator's execution. Each workspace stores three distinct pieces of state: `cur_ptr` and `scratch_pad` to track the iterator state described in §3, and `data`, which holds the data loaded from memory for `cur_ptr`. A dedicated workspace per iterator allows the logic pipeline to switch to any iterator's execution without delay when triggered by the scheduler, although it requires maintaining multiple workspaces — a maximum of $m + n$ to accommodate any possible schedule due to our bound on the number of concurrent iterators. We divide these workspaces equally across logic pipelines.

Memory pipeline: Each memory pipeline loads data from the attached DRAM to the corresponding workspace assigned by the scheduler at the start of each iteration. This involves (i) address translation and (ii) memory protection based on page access permissions. We realize range-based address translations (simulated in prior work [64]) in our real-world implementation using TCAM to reduce on-chip storage usage.

Once a memory access is complete, the memory pipeline signals the scheduler to continue the iterator execution or terminate it if there is a translation or protection failure.

Logic pipeline: Each logic pipeline runs PULSE ISA instructions other than `LOAD`/`STORE` to determine the `cur_ptr` value for the next iteration or, to determine if the termination condition has been met. Our logic pipeline comprises an ALU to execute the standard arithmetic and logic instructions, as well as modules to support register manipulation, branching, and the specialized `RETURN` instruction execution (Table 2). During a particular iterator's execution, the logic pipeline performs its corresponding instructions with direct reads and updates to its dedicated workspace registers. An iteration's logic can end in one of two possible ways: (i) the `cur_ptr` has been updated to the next pointer, and the `NEXT_ITER` instruction is reached, or (ii) the pointer traversal is complete, and the `RETURN` instruction is reached. In either case, the logic pipeline notifies the scheduler with the appropriate signal.

Scheduler: The scheduler handles new iterator requests received over the network and schedules each iterator's data fetch and logic execution across memory and logic pipelines:

1. On receiving a new request over the network, it assigns the iterator an empty workspace at a logic pipeline and signals one of the memory pipelines to execute the data fetch from memory based on the state in the workspace.
2. On receiving a signal from the memory pipeline that a data fetch has successfully completed, it notifies the appropriate logic pipeline to continue iterator execution via the corresponding workspace.
3. On receiving a signal from the logic pipeline that the next iteration can be started (via the `NEXT_ITER` instruction), it notifies one of the memory pipelines to execute `LOAD` via the corresponding workspace.
4. When it receives a signal from the memory pipeline that an address translation or memory protection failed or a signal from the logic pipeline that the iterator execution has met its terminal condition (via the `RETURN` instruction), it signals the network stack to prepare a response containing the iterator `code`, `cur_ptr` and `scratch_pad`.

While the scheduler assigns memory and logic pipelines to an iterator in steps 1 and 3 in a manner that maximizes utilization of all memory pipelines (i.e., Fig. 4 (bottom)), it is possible to implement other scheduling policies.

Network Stack: The network stack receives and transmits packets; when a new request arrives, it parses/deparses the payload to extract/embed the request ID, `code`, and state for the offloaded iterator execution (`cur_ptr`, `scratch_pad`).

The network stack uses the same format for both requests and responses, so a response can be sent back to the CPU node on traversal completion or rerouted as a request to a different memory node for continued execution (§5).

Implementation. We use an FPGA-based NIC (Xilinx Alveo U250) with two 100 Gbps ports, 64 GB on-board DRAM, 1,728K LUTs, and 70 Mb BRAM. Since the board has two Ethernet ports and four memory channels, we partition its resources into two PULSE accelerators, each with a single Ethernet port and two memory channels. Our analysis of common data structures (§6) shows their t_c/t_d ratio tends to be < 0.75. As such, we set $\eta = 0.75$, i.e., there are four memory and three logic pipelines and a total of 7 workspaces on the accelerator. We use the Xilinx TCAM IP [36] (for page tables), 100 Gbps Ethernet IP, link-layer IPs [153], and burst data transfers [8] to improve memory bandwidth. The logic and memory pipelines are clocked at 250 MHz, while the network stack operates at 322 MHz for 100 Gbps traffic. Our FPGA prototype showcases PULSE's potential; we believe that ASIC implementations are the next natural step.

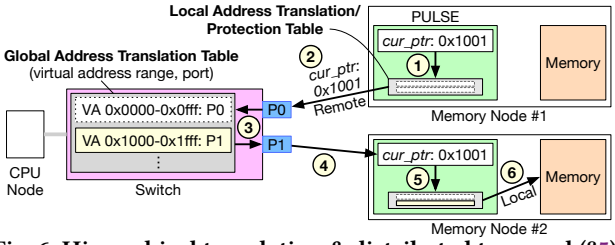


Fig. 6. Hierarchical translation & distributed traversal (§5).

5 Distributed Pointer Traversals

By restricting pointer traversals to a single memory node (§2), prior approaches leave applications with two undesirable options. At one extreme, they can confine their data to a single memory, but sacrifice application scalability. Conversely, they can spread their data across multiple nodes but have to return the CPU node whenever the traversal accesses a pointer on another memory node. This affords scalability but costs additional network and software processing latency at the CPU node. To avoid the cost, one may replicate the entire translation and protection state for the cluster at every memory node so they can directly forward traversal requests to other memory nodes. This comes at the cost of increased space consumption for translation, which is challenging to contain within the accelerator's translation and protection tables. Moreover, duplicating this state across memory nodes requires complex protocols for ensuring their consistency (*e.g.*, when the state changes), which have significant performance overheads.

PULSE breaks this tradeoff between performance and scalability by leveraging a programmable network switch to support rack-scale distributed pointer traversals. In particular, if the PULSE accelerator on one memory node detects that the next pointer lies on a different memory node, it forwards the request to the network switch, which routes it to the appropriate memory node for continuing the traversal. This cuts the network latency by half a round trip time and avoids software overheads at the CPU node, instead performing the routing logic in switch hardware. Since continuing the traversal across memory nodes is similar to packet routing, the switch hardware is already optimized to support it.

Enabling rack-scale pointer traversals, however, requires addressing two key challenges, as we discuss next.

Hierarchical translation. For the switch to forward the pointer traversal request to the appropriate memory node, it must be able to locate which memory nodes are responsible for which addresses. To minimize the logic and state maintained at the switch due to its limited resources, PULSE employs hierarchical address translation as shown in Fig. 6. In particular, the address space is range partitioned across memory nodes; PULSE only stores the base address to memory node mapping at the switch, while each memory node

stores its own local address translation and protection metadata at the accelerator (①), as outlined in §4. The routing logic at the switch inspects the `cur_ptr` field in the request (②) and consults its mapping to determine the target memory node (③). At the memory node, the traversal proceeds until the accessed pointer is not present in the local table (as in ①); it then sends the request back to the switch (§4.2), which can re-route the request to the appropriate memory node (④-⑥), or notify the CPU node if the pointer is invalid.

Continuing stateful iterator execution. One challenge of distributing iterator execution in PULSE lies in its stateful nature: since PULSE permits the storage of intermediate state in the iterator's `scratch_pad`, how can such stateful iterator execution be continued on a different memory node? Fortunately, our design choices of (i) confining all of the iterator state in `scratch_pad` and `cur_ptr` and (ii) keeping the request and response formats identical make this straightforward. The accelerator at the memory node simply embeds the up-to-date `scratch_pad` within the response before forwarding it to the switch; when the switch forwards it to the next memory node, it can simply continue execution exactly as it would have if the last memory node had the pointer.

6 Evaluation

Compared systems. We compare PULSE against: (i) a **Cache-based** system that relies solely on caches at CPU nodes to speed up remote memory accesses; we use Fastswap [42] as the representative system, (ii) an **RPC** system that offloads pointer-traversals to a CPU on memory nodes, (iii) **RPC-ARM**, an RPC system that employs a wimpy ARM processors at memory nodes, and (iv) a **Cache+RPC** approach that employs data structure-aware caches; we use AIFM [127] as the representative system. (i) and (iv) use a cache size of 2 GB, while (ii) and (iii) use a DPDK-based RPC framework [84].

Our experimental setup comprises two servers, one for the CPU node and the other for memory nodes, connected via a 32-port switch with a 6.4 Tbps programmable Tofino ASIC. Both servers were equipped with Intel Xeon Gold 6240 Processors [11] and 100 Gbps Mellanox ConnectX-5 NICs. For a fair comparison, we limit the memory bandwidth of the memory nodes to 25 GB/s (FPGA's peak bandwidth) using Intel Resource Director [27] and report energy consumption of the **minimum** number of CPU cores needed to saturate the bandwidth. We use Bluefield-2 [13] DPU as our ARM-based SmartNICs with 8 Cortex-A72 cores and 16 GB DRAM. For PULSE, we placed two memory nodes on each FPGA NIC (one per port, a total of 4 memory nodes). Our results translate to larger setups since PULSE's performance or energy efficiency are independent of dataset size and cluster scale.

Applications & workloads. We consider 3 applications with varying data structure complexity, compute/memory-access ratio, and iteration count per request (Table 3): (i) *Web Service* [127] that processes user requests by retrieving user IDs from an in-memory hash table, using these IDs to fetch 8 KB objects, which are then encrypted, compressed and returned to the user. Requests are generated using YCSB A (50% read/50% update), B (95% read/5% update), and C (100% read) workloads with Zipf distribution [58]. (ii) *WiredTiger Storage Engine* (MongoDB backend [108]) uses B+Trees to index NoSQL tables. Our frontend issues range query requests over the network to WiredTiger and plots the results. Similar to prior work [127, 164], we model user queries using the YCSB E workload with Zipf distribution [58] on 8 B keys and 240 B values. (iii) *BTrDB Time-series Database* [45] is a database designed for visualizing patterns in time-series data. BTrDB reads the data from a B+Tree-based store for a given user query and renders the time-series data through an interactive user interface [9]. We run stateful aggregations (sum, average, min, max) for time windows of different resolutions, from 1 s to 8 s, on the Open μPMU Dataset [137] with voltage, current, and phase readings from LBNL's power grid [45].

Application	Data Structure	t_c/t_d	#Iterations
WebService	Hash-table	0.06	48
WiredTiger	B+Tree	0.63	25
BTrDB (1s to 8s)		0.71	38–227

Table 3. Workloads used in our evaluation (§6). t_c and t_d correspond to compute and memory access time at the PULSE accelerator.

6.1 Performance for Real-world Applications

Since AIFM [127] does not natively support B+-Trees or distributed execution, we restrict the Cache+RPC approach to the Web Service application on a single node.

Single-node performance. Fig. 7 demonstrates the advantages of accelerating pointer-traversals at disaggregated memory. Compared to the Cache-based approach, PULSE achieves 9–34.4× lower latency and 28–171× higher throughput across all applications using only one network round-trip per request. RPC-based systems observe 1–1.4× lower latency than PULSE due to their 9× higher CPU clock rates. We believe an ASIC-based realization of PULSE has the potential to close or even overcome this gap. Cache+RPC incurs higher latency than RPC due to its TCP-based DPDK stack [117, 127] and does not outperform RPC, indicating that data structure-aware caching is not beneficial due to poor locality.

Latency depends on the number of nodes traversed during a single request and the response size. WebService experiences the highest latency due to large 8 KB responses and long traversal length per request. In BTrDB, the latency increases (and the throughput decreases) as the window size grows due to the longer pointer traversals (see Table 3). Interestingly, the Cache-based approach performs significantly better for BTrDB than WebService and WiredTiger due to the

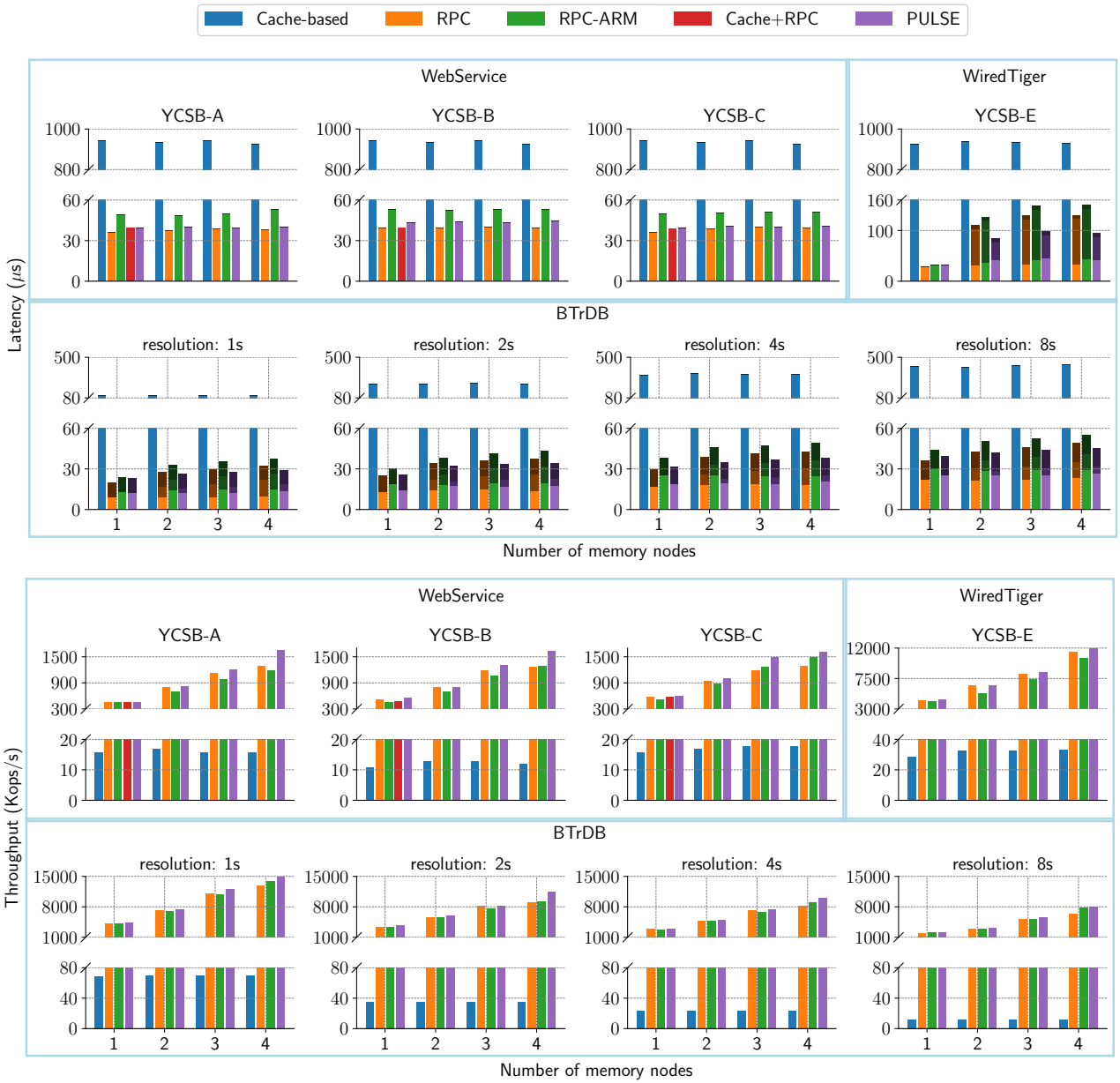


Fig. 7. Application latency (top) & throughput (bottom) (§6.1). The darker color indicates the time spent on cross-node pointer traversals, which increases with the number of memory nodes in WiredTiger and BTrDB.

better data locality in time-series analysis of chronologically ordered data. However, its throughput remains significantly lower than both PULSE and RPC since it is bottlenecked by the swap system performance, which could not evict pages fast enough to bring in new data. This is verified in our analysis of resource utilization (deferred to Appendix for brevity); we find that RPC, RPC-ARM, Cache+RPC, and PULSE can utilize more than 90% of the memory bandwidth across the applications, while the Cache-based approach observes less than 1 Gbps network bandwidth. The other systems — PULSE, RPC, RPC-ARM, and Cache+RPC — can also saturate available memory bandwidth (around 25 GB/s) by offloading pointer traversals to the memory node, consuming only 0.5%–25% of the available network bandwidth.

Distributed pointer traversals. Fig. 7 shows that employing multiple memory nodes introduces two major changes in performance trends: (i) the latency increases when the pointer traversal spans multiple memory nodes, and (ii) throughput increases with the number of nodes since the systems can exploit more CPUs or accelerators. WebService is an exception to the trend: since the hash table is partitioned across memory nodes based on primary keys, the linked list for a hash bucket resides in a single memory node.

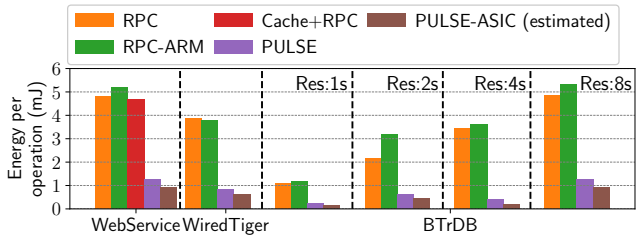


Fig. 8. Application energy consumption per operation (§6.1).

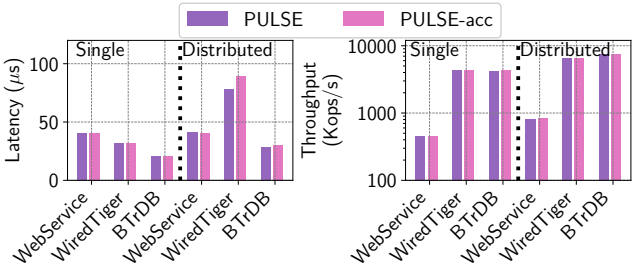


Fig. 9. Impact of distributed pointer traversals (§6.2).

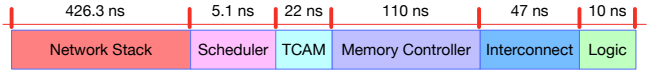


Fig. 10. Latency breakdown for PULSE accelerator (§6.2).

	#Logic Pipelines	#Memory Pipelines	LUT %	BRAM %	Throughput (Mops/s)	Latency (us)
Coupled	1	1	7.37	7.29	0.41	33.25
	2	2	10.23	9.37	0.63 (+53%)	33.73
	3	3	14.33	15.92	0.87 (+112%)	34.66
	4	4	18.55	17.09	1.20 (+193%)	35.11
PULSE	1	1	5.88	8.17	0.51	37.57
	1	2	7.44	9.14	0.73 (+43%)	36.74
	1	3	8.32	11.19	1.01 (+98%)	38.46
	1	**4**	**9.19**	**12.92**	**1.24 (+143%)**	**38.37**
	2	1	8.87	10.19	0.48 (-6%)	40.27
	2	2	10.69	11.19	0.76 (+49%)	39.47
	2	3	13.11	13.38	0.99 (+94%)	41.37
	2	4	15.07	15.61	1.19 (+133%)	40.37
	3	1	14.08	11.93	0.46 (-10%)	42.38
	3	2	15.79	13.78	0.69 (+35%)	43.11
	3	3	18.61	15.06	1.03 (+102%)	40.98
	3	4	19.20	17.47	1.17 (+129%)	44.02
	4	1	18.67	14.17	0.37 (-27%)	42.16
	4	2	20.37	16.02	0.51 (0%)	43.00
	4	3	22.08	17.86	1.10 (+116%)	43.86
	4	4	23.21	19.92	1.14 (+123%)	41.47

Table 4. Coupled (multi-core) vs. PULSE's disaggregated architecture (§6.2). The highlighted configuration depicts PULSE's Pareto-optimal resource usage and performance.

PULSE observes lower latency than the compared systems due to in-network support for distributed pointer-traversals (§5). The latency increases significantly from one to two memory nodes for all systems since traversing to the next pointer on a different memory node adds 5–10 μs network latency. Also, even across two memory nodes, a request can trigger multiple inter-node pointer traversals incurring multiple network round-trips; for WiredTiger and BtrDB, 10%–30% of pointer traversals are inter-node. However, in-network traversals allow PULSE to reduce latency overheads by 33–98%, with 1.1–1.36× higher throughput than RPC.

Energy consumption. We compared energy consumed per request for PULSE and RPC schemes at a request rate that ensured memory bandwidth was saturated for both. We measure energy consumption using Xilinx XRT [37] for PULSE (all power rails) and Intel RAPL tools [78] for RPC on CPUs [11] (CPU package and DRAM only). For RPC-ARM on ARM cores, since there is no power-related performance counter [17] or open-source tool available, we adapt the measurement approach from prior work [74]. Specifically, we calculate the CPU package's energy using application CPU cycle counts and DRAM power using Micron's estimation tool [25]. Finally, we conservatively estimate ASIC power using our FPGA prototype: we scale down the ASIC energy only for PULSE accelerator using the methodology employed in prior research [95] while using the unscaled FPGA energy for other components (DRAM, third-party IPs, etc.). As such, we measure an *upper bound* on PULSE and PULSE-ASIC energy use, and a *lower bound* for RPC, RPC-ARM, and Cache+RPC.

Fig. 8 shows that PULSE achieves a 4.5–5× reduction in energy use per operation compared to RPCs on a general-purpose CPU, due to its disaggregated architecture (§4.2). Our estimation shows that PULSE's ASIC realization can conservatively reduce energy use by an additional 6.3−7× factor. Finally, RPC-ARM's total energy consumption per request can exceed that of standard cores, as seen in the WebService workload. This observation aligns with prior studies [74], which attribute the increased energy use to their longer execution times, resulting in higher aggregate energy demands.

6.2 Understanding PULSE Performance

Distributed pointer traversals. We evaluate the impact of distributed pointer traversals (§5) by comparing PULSE against PULSE-ACC, a PULSE variant that sends requests back to the CPU node if the next pointer is not found on the memory node. Fig. 9 shows that while both have identical performance on a single memory node, PULSE-ACC observes 1.02–1.15× higher latency for two nodes. On the other hand, their throughput is the same since, under sufficient load, memory node bandwidth bottlenecks the system for both.

Latency breakdown for PULSE accelerator. Fig. 10 shows the latency contributions of various hardware components at the PULSE accelerator for the WebService application. The network stack first processes the pointer traversal request in about 430 ns, after which the WebService payload is processed by the scheduler and dispatched to an idle memory access pipeline in 5.1 ns. Then, the memory pipeline takes ~132 ns to perform address translation, memory protection, and data fetch from DRAM. Finally, the logic pipeline takes 10 ns to check the termination conditions and determine the next pointer to look up. This process repeats until the termination condition is met. The time to send a response back over the network stack is symmetric to the request path.

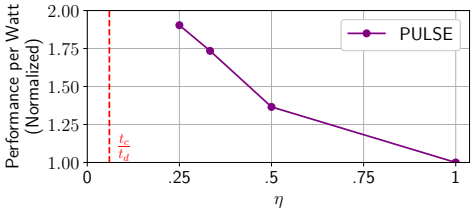


Fig. 11. PULSE sensitivity to η (§6.2). Performance-per-watt is normalized by the value at $\eta = 1$.

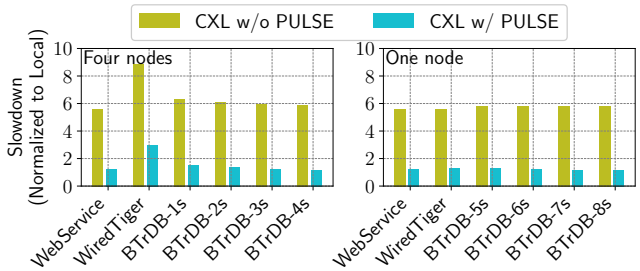


Fig. 12. Slowdown with simulated CXL interconnect (§7).

Benefits of disaggregating memory and logic pipelines. Table 4 compares the area usage (percentage of BRAM/LUT resources used on FPGA) and performance (throughput/latency for the WebService application) of PULSE's disaggregated and traditional coupled (i.e., multi-core) designs, which combines logic and memory pipelines into cores. PULSE requires slightly more area than the coupled design when the number of logic and memory pipelines are equal to accommodate additional logic and buffers across the interconnect and the scheduler. However, due to the memory-intensive nature of pointer traversal operations (§4), PULSE can achieve similar performance with fewer logic pipelines and, therefore, less area. To saturate memory bandwidth (and thus maximize throughput) for the WebService application, PULSE only needs one logic pipeline and four memory pipelines, while a traditional core architecture must use four cores. As such, PULSE saves 38% area with a marginal 7% latency increase due to scheduling and workspace overheads.

Sensitivity to η Parameter. We evaluate PULSE 's sensitivity to η by varying the number of memory pipelines with a single logic pipeline for the WebService application. Figure 11 shows that as η in PULSE accelerator approaches the workload's compute-to-memory ratio ($\sim$ 1/16), its performance per watt improves significantly since the accelerator resources better match the workload needs. In contrast, large η values lead to underutilization of the logic pipeline and, thus, wasted energy. For instance, decreasing the η value from 1 to 1/4 increases the performance-per-watt by 1.9×!

7 Future Trends and Research

Next-generation interconnects. While PULSE is implemented atop Ethernet, its design is interconnect-agnostic. It could be realized over emerging interconnects like CXL [24, 102, 138]. We have verified these benefits in simulation atop traces of our evaluated applications. Our simulator used 2 GB DRAM as CPU-attached cache, while the entire working set is stored on CXL-attached memory. Following prior work [101], we model 10–20 ns L3 cache latency, 80 ns DRAM latency, 300 ns CXL-attached memory latency, and 256 B access granularity. We simulate both a four-memory-node setup, which uses a CXL switch with PULSE logic and a PULSE accelerator at each memory node, and a single-node setup with no switch. We assume a conservative overhead for PULSE, using our Ethernet switch and FPGA latencies.

Fig. 12 shows the slowdown of workloads on CXL memory versus local DRAM, both with and without PULSE. PULSE reduces CXL's slowdown by 3–5× in the four-node setup, and by 4.2–5.2× in the single-node setup. While real hardware realization is necessary to quantify PULSE's benefits precisely, our simulation (with optimistic CXL latency and conservative PULSE overheads) highlights the potential for improving performance in such interconnects.

Data encryption in memory. With increasing focus on trusted server infrastructure for secure cloud, an interesting avenue of future research is enabling near-memory processing over encrypted disaggregated memory. We identify two critical challenges. The first involves managing encryption keys securely, especially since the PULSE accelerator, an intermediary, could be compromised. We argue for incorporating a Trusted Execution Environment (TEE) in PULSE, similar to prior FPGA systems that isolate sensitive key management functions [114, 119, 126, 150, 163]. The second challenge involves hiding memory access patterns as a side channel over encrypted memory [55, 79, 88, 91]. While several recent advances in noise injection techniques permit efficient defense mechanisms against side-channel attacks [71, 82, 143], developing performant solutions in hardware remains an open problem.

8 Related work

Memory disaggregation. Disaggregated memory systems span both RDMA-based [42, 72, 100, 130] and CXL-based interconnects [69, 70, 140, 147]. Even with gigabytes of DRAM at compute nodes, these approaches observe significant performance degradation for workloads with poor data locality when most data accesses hit slower disaggregated memory. Application-integrated disaggregated memory schemes alleviate some of the performance overheads for specific scenarios, *e.g.*, garbage collection, key-value storage, etc. [127, 141, 144, 145], but do not generalize to other scenarios. PULSE aims to enable efficient execution for a large class of pointer traversal workloads by placing general but lightweight processing primitives close to the memory nodes.

Near-memory processing. Limited data bandwidth between compute and storage devices and the high cost of

data movement are well-documented in the architecture community. Several works have proposed hardware architectures that move computations close to storage or memory [52, 57, 61, 148], albeit with limited flexibility for expressing offloaded logic. Another related class of approaches has targeted graph-processing accelerators for machine learning workloads [86, 96] — these still suffer from limited expressiveness for general data structures. Recent efforts in the industry have also explored placing accelerators at or close to memory devices [83, 125]. Unfortunately, most approaches require micro-architectural modifications to enable near- or on-memory processing. PULSE instead focuses on leveraging programmable networks for accelerating pointer traversals on linked structures in disaggregated architectures.

Prefetching. As noted in §2.2, while prefetching [41, 103, 155] can be used to pipeline remote memory accesses during pointer traversals over disaggregated memory, its benefits are limited in for pointer traversals. However, prefetching and PULSE's approach of near-memory processing are orthogonal. Indeed, PULSE *complements* prefetching by enhancing performance for workloads where the effectiveness of prefetching techniques is limited.

9 Conclusion

We have designed PULSE to accelerate pointer traversals across linked data structures close to disaggregated memory in a manner that preserves expressiveness, ensures energy efficiency, and supports distributed execution. PULSE makes a principled use of near-memory acceleration, and programmable network switches for low-latency, high-throughput pointer traversals on disaggregated memory.

Acknowledgements

We would like to thank our shepherd Zsolt István and anonymous ASPLOS reviewers for their valuable comments and insightful feedback. This work is supported in part by NSF Awards 2047220, 2112562, 2147946, 2118851, and a NetApp Faculty Fellowship.

References

[1] 2007. Boost AVL tree. https://www.boost.org/doc/libs/1_35_0/doc/html/intrusive/avl_set_multiset.html.
[2] 2007. Boost library. https://www.boost.org/.
[3] 2007. Boost splay tree. https://www.boost.org/doc/libs/1_35_0/doc/html/intrusive/splay_set_multiset.html.
[4] 2008. Boost scapegoat tree. https://www.boost.org/doc/libs/1_38_0/doc/html/intrusive/sg_set_multiset.html.
[5] 2008. Boost unordered map. https://www.boost.org/doc/libs/1_38_0/doc/html/boost/unordered_map.html.
[6] 2008. Boost unordered set. https://www.boost.org/doc/libs/1_51_0/doc/html/boost/unordered_set.html.
[7] 2011. Google BTree. https://code.google.com/archive/p/cpp-btree/.
[8] 2013. AMBA AXI and ACE Protocol Specification. https://developer.arm.com/documentation/ihi0022/e/?lang=en.
[9] 2016. Mr. Plotter: A Multi-Resolution Plotter compatible with BTrDB. https://github.com/BTrDB/mr-plotter.
[10] 2017. RoCE vs. iWARP Competitive Analysis. https://www.mellanox.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf.
[11] 2019. Intel Xeon Gold 6240 Processor datasheet. https://ark.intel.com/content/www/us/en/ark/products/192443/intel-xeon-gold-6240-processor-24-75m-cache-2-60-ghz.html.
[12] 2019. Terabit Ethernet: The New Hot Trend in Data Centers. https://www.lanner-america.com/blog/terabit-ethernet-new-hot-trend-data-centers/.
[13] 2020. NIVIDIA MELLANOX BLUEFIELD-2. https://network.nvidia.com/files/doc-2020/pb-bluefield-2-smart-nic-eth.pdf.
[14] 2022. Boost bimap. https://www.boost.org/doc/libs/1_80_0/libs/bimap/doc/html/index.html.
[15] 2023. C++ std::iterator. https://en.cppreference.com/w/cpp/iterator/iterator.
[16] 2023. Standard containers. https://cplusplus.com/reference/stl/.
[17] 2024. AArch64 Performance Monitors registers. https://developer.arm.com/documentation/100095/0002/system-control/aarch64-register-summary/aarch64-performance-monitors-registers.
[18] 2024. C++ standard forward_list container. https://en.cppreference.com/w/cpp/container/forward_list.
[19] 2024. C++ standard list container. https://en.cppreference.com/w/cpp/container/list.
[20] 2024. C++ standard map container. https://en.cppreference.com/w/cpp/container/map.
[21] 2024. C++ standard multimap container. https://en.cppreference.com/w/cpp/container/multimap.
[22] 2024. C++ standard multiset container. https://en.cppreference.com/w/cpp/container/multiset.
[23] 2024. C++ standard set container. https://en.cppreference.com/w/cpp/container/set.
[24] 2024. Compute Express Link (CXL). https://www.computeexpresslink.org/.
[25] 2024. DDR4 POWER CALC.XLSM. https://www.micron.com/sales-support/design-tools/dram-power-calculator.
[26] 2024. DPDK. https://www.dpdk.org/.
[27] 2024. Intel(R) RDT Software Package. https://github.com/intel/intel-cmt-cat.
[28] 2024. Java iterator. https://www.w3schools.com/java/java_iterator.asp.
[29] 2024. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html#introduction.
[30] 2024. MemCached. http://www.memcached.org.
[31] 2024. MySQL: Adaptive Hash Index. https://dev.mysql.com/doc/refman/8.0/en/innodb-adaptive-hash.html.
[32] 2024. Teradata: Hash Indexes. https://docs.teradata.com/r/Enterprise_IntelliFlex_VMware/Database-Design/Join-and-Hash-Indexes/Hash-Indexes.
[33] 2024. The LLVM Compiler Infrastructure. https://llvm.org/.
[34] 2024. VoltDB. http://voltdb.com/downloads/datasheets_collateral/technical_overview.pdf.
[35] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. 2013. Scuba: Diving into Data at Facebook. *PVLDB* 6, 11 (2013).

[36] Inc. Advanced Micro Devices. 2024. Xilinx Content Addressable Memory (CAM). https://www.xilinx.com/products/intellectual-property/ef-di-cam.html.

[37] Inc. Advanced Micro Devices. 2024. Xilinx Runtime Library (XRT). https://www.xilinx.com/products/design-tools/vitis/xrt.html.

[38] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling Queries on Compressed Data. In *USENIX NSDI*.

[39] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote Memory in the Age of Fast Networks. In *SoCC*.

[40] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*. 105–117.

[41] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively prefetching remote memory with leap. In *USENIX ATC*. 843–857.

[42] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *EuroSys*.

[43] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Remote Memory Calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 38–44.

[44] Hang An, Fang Wang, Dan Feng, Xiaomin Zou, Zefeng Liu, and Jianshun Zhang. 2023. Marlin: A Concurrent and Write-Optimized B+-tree Index on Disaggregated Memory. In *ACM ICPP*.

[45] Michael P. Andersen and David E. Culler. 2016. BTrDB: Optimizing Storage System Design for Timeseries Processing. In *USENIX FAST*.

[46] Krste Asanović. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers.

[47] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. 2016. Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems. In *IEEE/ACM MICRO*.

[48] Nikolas Askitis and Ranjan Sinha. 2007. HAT-trie: A Cache-conscious Trie-based Data Structure for Strings. In *ACSC*.

[49] R. Bayer and E. McCreight. 1970. Organization and Maintenance of Large Ordered Indices. In *ACM-SIGMOD Workshop on Data Description, Access and Control*.

[50] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *USENIX OSDI*.

[51] Ankit Bhardwaj, Chinmay Kulkarni, and Ryan Stutsman. 2020. Adaptive Placement for In-memory Storage Functions. In *USENIX ATC*.

[52] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. 2019. CoNDA: Efficient Cache Coherence Support for near-Data Accelerators. In *ISCA*. 629–642.

[53] Anastasia Braginsky and Erez Petrank. 2012. A Lock-free B+Tree. In *SPAA*.

[54] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX ATC*.

[55] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *ACM CCS*.

[56] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 27–39.

[57] Benjamin Y. Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez. 2020. Near Data Acceleration with Concurrent Host Access. In *ISCA*. 818–831.

[58] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. 143–154.

[59] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2018. GraphH: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (2018), 640–653.

[60] Guohao Dai, Zhenhua Zhu, Tianyu Fu, Chiyue Wei, Bangyan Wang, Xiangyu Li, Yuan Xie, Huazhong Yang, and Yu Wang. 2022. Dimming: pruning-efficient and parallel graph mining on near-memory-computing. In *ISCA*. 130–145.

[61] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. 2022. To PIM or Not for Emerging General Purpose Processing in DDR Memory Systems. In *ISCA*. 231–244.

[62] Charles Eckert, Arun Subramaniyan, Xiaowei Wang, Charles Augustine, Ravishankar Iyer, and Reetuparna Das. 2022. Eidetic: An in-memory matrix multiplication accelerator for neural networks. *IEEE Trans. Comput.* (2022).

[63] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *USENIX NSDI*.

[64] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Ünsal. 2016. Range Translations for Fast Virtual Memory. *IEEE Micro* 36, 3 (2016), 118–126.

[65] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *USENIX OSDI*.

[66] Juan Gómez-Luna, Yuxin Guo, Sylvan Brocard, Julien Legriel, Remy Cimadomo, Geraldo F Oliveira, Gagandeep Singh, and Onur Mutlu. 2023. Evaluating machine learning workloads on memory-centric computing systems. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 35–49.

[67] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX OSDI*.

[68] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *USENIX OSDI*.

[69] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory pooling with cxl. *IEEE Micro* 43, 2 (2023), 48–57.

[70] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct access, High-Performance memory disaggregation with DirectCXL. In *USENIX ATC*.

[71] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. 2020. Pancake: Frequency smoothing for encrypted data stores. In *USENIX Security*.

[72] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *USENIX NSDI*.

[73] Peng Gu, Xinfeng Xie, Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, and Yuan Xie. 2020. iPIM: Programmable in-memory image processing accelerator using near-bank architecture. In *ISCA*. IEEE, 804–817.

[74] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *ACM ASPLOS*.

[75] Steffen Heinz, Justin Zobel, and Hugh E Williams. 2002. Burst tries: a fast, efficient data structure for string keys. *TOIS* (2002).

[76] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K. Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *International Conference on Computer Design (ICCD)*.

[77] Stratos Idreos, F. Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35 (01 2012).

[78] Intel Corporation. 2024. *Intel 64 and IA-32 Architectures Software Developer's Manual*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[79] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*.

[80] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison cache: A scalable and effective die-stacked DRAM cache. In *IEEE/ACM MICRO*.

[81] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 404–415.

[82] Grace Jia, Rachit Agarwal, and Anurag Khandelwal. 2024. Length Leakage in Oblivious Data Access Mechanisms. In *USENIX Security*.

[83] Dave Jiang. 2019. Introducing the Intel® Data Streaming Accelerator (Intel® DSA). https://01.org/blogs/2019/introducing-intel-data-streaming-accelerator.

[84] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *USENIX NSDI*.

[85] Uksong Kang, Hak-Soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, S Jang, and Joo Sun Choi. 2014. Co-architecting controllers and DRAM to enhance DRAM process scaling. In *The memory forum*, Vol. 14.

[86] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. RecNMP: Accelerating Personalized Recommendation with near-Memory Processing. In *ISCA*. 790–803.

[87] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, et al. 2021. Near-memory processing in action: Accelerating personalized recommendation with axdimm. *IEEE Micro* 42, 1 (2021), 116–127.

[88] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *ACM CCS*.

[89] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. 2016. BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores.. In *USENIX NSDI*.

[90] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for in-Memory Databases. In *IEEE/ACM MICRO*.

[91] Evgenios Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2019. Data Recovery on Encrypted Databases with k-Nearest Neighbor Query Leakage. In *IEEE S&P*.

[92] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *USENIX OSDI*.

[93] Konstantinos Koukos, David Black-Schaffer, Vasileios Spiliopoulos, and Stefanos Kaxiras. 2013. Towards More Efficient Execution: A Decoupled Access-Execute Approach. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*.

[94] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. 2018. Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage. In *USENIX OSDI*.

[95] Ian Kuon and Jonathan Rose. 2006. Measuring the Gap between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays (FPGA '06)*. 21–30.

[96] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. TensorDIMM: A Practical near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *IEEE/ACM MICRO*. 740–753.

[97] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *USENIX OSDI*.

[98] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO 2004)*. 75–86.

[99] Seok-Hee Lee. 2016. Technology scaling challenges and opportunities of memory devices. In *International Electron Devices Meeting (IEDM)*.

[100] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *SOSP*.

[101] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *ACM ASPLOS*.

[102] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, et al. 2022. First-generation Memory Disaggregation for Cloud Platforms. *arXiv preprint arXiv:2203.00241* (2022).

[103] Haifeng Li, Ke Liu, Ting Liang, Zuojun Li, Tianyue Lu, Hui Yuan, Yinben Xia, Yungang Bao, Mingyu Chen, and Yizhou Shan. 2023. HoPP: Hardware-Software Co-Designed Page Prefetching for Disaggregated Memory. In *IEEE HPCA*.

[104] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *USENIX FAST*.

[105] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. 2020. Livia: Data-centric computing throughout the memory hierarchy. In *ACM ASPLOS*.

[106] Microsoft Corporation. 2024. *SQL Server and Azure SQL Index Architecture and Design Guide*. https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide?view=sql-server-ver16#hash_index.

[107] Xinhao Min, Kai Lu, Pengyu Liu, Jiguang Wan, Changsheng Xie, Daohui Wang, Ting Yao, and Huatao Wu. 2024. SepHash: A Write-Optimized Hash Index On Disaggregated Memory via Separate Segment Structure. *Proc. VLDB Endow.* 17, 5 (2024), 1091–1104.

[108] Inc. MongoDB. 2024. WiredTiger storage engine. https://docs.mongodb.com/manual/core/wiredtiger/.

[109] Donald R. Morrison. 1968. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *JACM* (1968).

[110] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2019. Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems* 67 (2019), 28–41.

[111] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2022. A modern primer on processing in memory. In *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*. Springer, 171–243.

[112] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *USENIX NSDI*.

[113] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. 2019. Storm: A Fast Transactional Dataplane for Remote Data Structures. In *SYSTOR*. 97–108.

[114] Hyunyoung Oh, Kevin Nam, Seongil Jeon, Yeongpil Cho, and Yunheung Paek. 2021. MeetGo: A trusted execution environment for remote applications on FPGA. *IEEE Access* 9 (2021), 51313–51324.

[115] Ataberk Olgun, Juan Gómez Luna, Konstantinos Kanellopoulos, Behzad Salami, Hasan Hassan, Oguz Ergin, and Onur Mutlu. 2022. PiDRAM: A Holistic End-to-end FPGA-based Framework for Processing-in-DRAM. *ACM Transactions on Architecture and Code Optimization* 20, 1 (2022), 1–31.

[116] Geraldo F Oliveira, Juan Gómez-Luna, Saugata Ghose, Amirali Boroumand, and Onur Mutlu. 2022. Accelerating neural network inference with processing-in-DRAM: from the edge to the cloud. *IEEE Micro* 42, 6 (2022), 25–38.

[117] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI*.

[118] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report.

[119] Sérgio Pereira, David Cerdeira, Cristiano Rodrigues, and Sandro Pinto. 2021. Towards a trusted execution environment via reconfigurable FPGA. *arXiv preprint arXiv:2107.03781* (2021).

[120] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. 2022. RDMA is Turing complete, we just did not know it yet!. In *USENIX NSDI*.

[121] Redis. 2024. Redis - The Real-time Data Platform. https://redis.io/.

[122] Charles Reiss. 2016. *Understanding Memory Configurations for In-Memory Analytics*. Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-136.html

[123] John C Reynolds. 1993. The discoveries of continuations. *Lisp and symbolic computation* 6 (1993), 233–247.

[124] Alessandro Rivitti, Roberto Bifulco, Angelo Tulumello, Marco Bonola, and Salvatore Pontarelli. 2023. eHDL: Turning eBPF/XDP Programs into Hardware Designs for the NIC. In *ACM ASPLOS*.

[125] Daniel Robinson. 2021. Samsung to Bring In-Memory Processing to Standard DIMMs and Mobile Memory. https://blocksandfiles.com/2021/08/24/samsung-to-bring-in-memory-processing-to-standard-dimms-and-mobile-memory/.

[126] Paul D. Rosero-Montalvo, Zsolt István, and Wilmar Hernandez. 2023. A Survey of Trusted Computing Solutions Using FPGAs. *IEEE Access* 11 (2023), 31583–31593.

[127] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *USENIX OSDI*.

[128] Fabian Schuiki, Michael Schaffner, Frank K Gürkaynak, and Luca Benini. 2018. A scalable near-memory architecture for training deep neural networks on large in-memory datasets. *IEEE Trans. Comput.* 68, 4 (2018), 484–497.

[129] Vivek Seshadri and Onur Mutlu. 2017. Simple operations in memory to reduce data movement. In *Advances in Computers*. Vol. 106. Elsevier, 107–166.

[130] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *USENIX OSDI*.

[131] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R. Lyu. 2023. Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System. In *SOSP*.

[132] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *USENIX FAST*.

[133] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. 2020. FlightTracker: Consistency across Read-Optimized Online Stores at Facebook. In *USENIX OSDI*.

[134] Shigeru Shiratake. 2020. Scaling and Performance Challenges of Future DRAM. In *International Memory Workshop (IMW)*.

[135] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: Smart Remote Memory. In *EuroSys*.

[136] Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Dionysios Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu. 2021. FPGA-based near-memory acceleration of modern data-intensive applications. *IEEE Micro* 41, 4 (2021), 39–48.

[137] Emma M. Stewart, Anna Liao, and Ciaran Roberts. 2016. Open μPMU: A Real World Reference Distribution Micro-phasor Measurement Unit Data Set for Research and Application Development. (2016).

[138] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices.

[139] Yupeng Tang, Seung-seob Lee, Abhishek Bhattacharjee, and Anurag Khandelwal. 2023. PULSE: Accelerating Distributed Pointer-Traversals on Disaggregated Memory. arXiv:2305.02388 [cs.DC] https://arxiv.org/abs/2305.02388.

[140] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *EuroSys*.

[141] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *USENIX ATC*.

[142] Fengbin Tu, Yiqi Wang, Zihan Wu, Ling Liang, Yufei Ding, Bongjin Kim, Leibo Liu, Shaojun Wei, Yuan Xie, and Shouyi Yin. 2022. ReDCIM: Reconfigurable digital computing-in-memory processor with unified FP/INT pipeline for cloud AI acceleration. *IEEE Journal of Solid-State Circuits* 58, 1 (2022), 243–255.

[143] Midhul Vuppalapati, Kushal Babel, Anurag Khandelwal, and Rachit Agarwal. 2022. SHORTSTACK: Distributed, Fault-tolerant, Oblivious Data Access. In *USENIX OSDI*.

[144] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A Memory-Disaggregated Managed Runtime. In *USENIX OSDI*.

[145] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime. In *USENIX OSDI*.

[146] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *SIGMOD*.

[147] Zhonghua Wang, Yixing Guo, Kai Lu, Jiguang Wan, Daohui Wang, Ting Yao, and Huatao Wu. 2024. Rcmp: Reconstructing RDMA-Based Memory Disaggregation via CXL. *ACM Transactions on Architecture and Code Optimization* 21, 1 (2024), 1–26.

[148] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. 2022. Near-Stream Computing: General and Transparent near-Cache Acceleration. In *HPCA*. 331–345.

[149] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony Nowatzki. 2021. Stream floating: Enabling proactive and decentralized cache optimizations. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 640–653.

[150] Ke Xia, Yukui Luo, Xiaolin Xu, and Sheng Wei. 2021. Sgx-fpga: Trusted execution environment for cpu-fpga heterogeneous architecture. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 301–306.

[151] Xinfeng Xie, Peng Gu, Yufei Ding, Dimin Niu, Hongzhong Zheng, and Yuan Xie. 2023. MPU: Memory-centric SIMT Processor via In-DRAM Near-bank Computing. *ACM Transactions on Architecture and Code Optimization* 20, 3 (2023), 1–26.

[152] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 570–583.

[153] Xilinx. 2022. XUP Vitis Network Example (VNx). https://github.com/Xilinx/xup_vitis_network_example.

[154] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *USENIX OSDI*.

[155] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. 2023. DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation. In *EuroSys*.

[156] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. 2021. Ship Compute or Ship Data? Why Not Both?. In *USENIX NSDI*. 633–651.

[157] Vinson Young, Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. 2018. Accord: Enabling associativity for gigascale dram caches by coordinating way-install and way-prediction. In *ISCA*.

[158] Xiangyao Yu, George Bezerra, Andrew Pavlo, Sahana Devadas, and Michael Stonebraker. 2014. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment* 8 (11 2014).

[159] Zhuolong Yu, Yiwen Zhang, Vladimir Bravermann, Mosharaf Chowdhury, and Xin Jin. 2009. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *SIGCOMM*.

[160] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX NSDI*.

[161] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *SIGMOD*.

[162] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2022. Optimizing Data-Intensive Systems in Disaggregated Data Centers with TELEPORT. In *SIGMOD*.

[163] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. 2022. ShEF: shielded enclaves for cloud FPGAs. In *ACM ASPLOS*.

[164] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *USENIX OSDI*.