Abstract

Optimizing Memory Management for Disaggregated Architectures

Yupeng Tang

2024

The increasing demand for scalable and efficient data center architectures has led to the adoption of resource disaggregation, which separates compute, memory, and storage resources across various interconnects. This paradigm shift from traditional monolithic server architectures allows for more flexible resource allocation and utilization. Memory disaggregation, in particular, addresses the bottleneck issues of traditional setups by decoupling memory resources, presenting them as pooled resources accessible on demand. This approach enhances efficiency, scalability, and adaptability, especially for memory-intensive workloads.

However, transitioning existing applications to a disaggregated architecture presents significant challenges due to the mismatch between current cloud stacks designed for monolithic systems and the requirements of disaggregated systems. These challenges span across different layers of the stack, including application interfaces, OS support, performance overheads, and the limitations of existing interconnect technologies. This dissertation focuses on addressing these challenges, particularly in the context of memory management within disaggregated architectures.

Our approach involves a comprehensive examination of the requirements for successful disaggregation, proposing strategies to mitigate performance penalties and enhance resource management. By adopting a top-down perspective, we aim to bridge the gap between service layers and core hardware elements, ultimately facilitating the transition to disaggregated data center architectures.

Optimizing Memory Management for Disaggregated Architectures

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Yupeng Tang

Dissertation Director: Anurag Khandelwal

Dec, 2024

Contents

List of Figures

List of Tables

Acknowledgements

A lot of people are awesome. Probably your family, friends, advisor, and that one super special high school teacher who believed in you.

Chapter 1

Introduction

The growing demand for scalable and efficient data center architectures has led to the emergence of resource disaggregation [1–9]. This modern paradigm represents a significant shift from traditional monolithic server architectures. In conventional setups, servers are typically equipped with a fixed combination of compute, memory, and storage resources. In contrast, resource-disaggregated systems physically separate these resources and distribute them across various interconnects, such as networks [1–3], CXL [10, 11], and others. This separation allows for more flexible resource allocation and utilization.

Within the broader context of resource disaggregation in modern data center architectures, **memory disaggregation** [4–9] plays a crucial and foundational role. In traditional monolithic architecture, memory often becomes a bottleneck, limiting the scalability and adaptability of applications. This issue has been frequently observed and reported in production data centers [12–21]. By decoupling memory resources from compute and storage elements and presenting them as pooled, disaggregated resources [22, 23], data centers can achieve increased efficiency, scalability, and adaptability. Memory-intensive applications [24–26] can access the memory they need on demand, without being constrained by the limitations of individual servers. Memory disaggregation is the first step toward realizing the full potential of resource disaggregation, enabling data centers to efficiently allocate and utilize resources based on dynamic application needs.

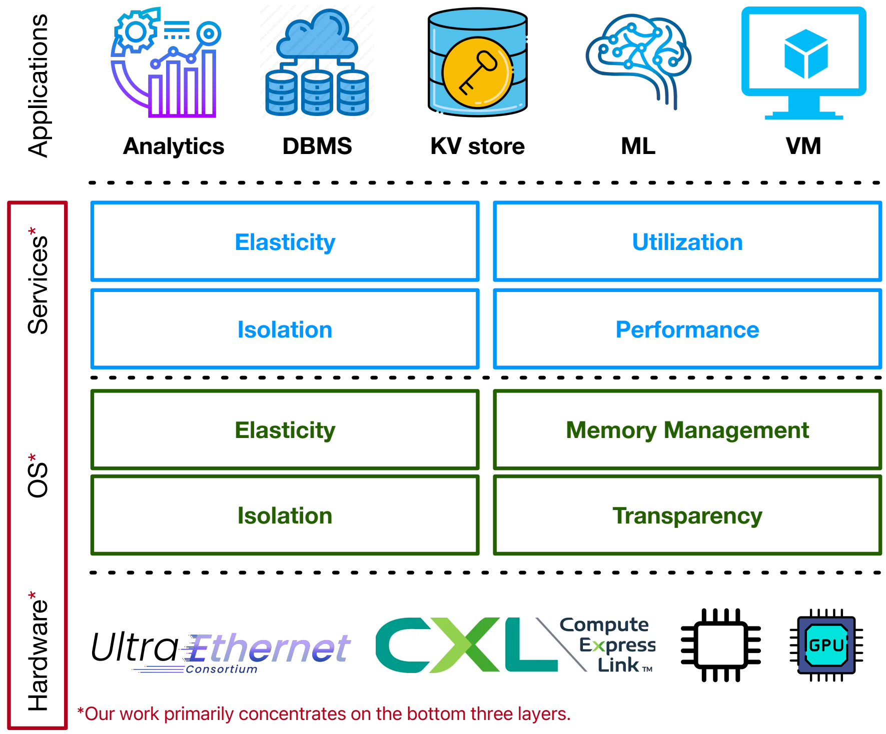


Fig. 1.1: **Cloud Stack of Disaggregated Architecture.**

1.1 Limitations of Existing Approaches

While memory disaggregation offers numerous advantages, transitioning existing applications to this architecture is far from straightforward. Recent research has explored various approaches to tackle this challenge. Some efforts focus on optimizing applications for disaggregated memory [27–30], while others aim to transparently port applications, offloading the responsibility for mitigating performance penalties—stemming from mismatches between disaggregated architectures and traditional software interfaces—to the service or operating system layer [1, 2, 31–34]. Meanwhile, hardware innovations further complicate the situation. Different interconnects, such as Ethernet and CXL, offer diverse interfaces and performance characteristics, making it difficult to standardize software stack designs.

The core issue lies in the fundamental mismatch between the existing cloud stack, designed for monolithic architectures, and the requirements of disaggregated architectures (Figure 1.1). The current cloud and hardware stacks are not inherently aware of the unique characteristics of disaggregated memory, leading to distinct challenges across different layers of the stack:

Application interface. In disaggregated architectures, applications encounter unique challenges compared to traditional monolithic systems. The key difference lies in resource distribution: compute, memory, and storage are spread across multiple resource nodes rather than centralized within a single server. This distribution necessitates complex communication and data management strategies to address increased latency and resource coordination. Transparency also becomes a critical concern. There is an inherent trade-off between the benefits of disaggregating resources and the associated performance overhead. Application developers may either significantly redesign their applications for more efficient resource utilization and management or rely on disaggregated cloud providers to integrate resource management at the service or operating system layer to ensure a smoother transition.

Operating system support. Unlike monolithic servers where the operating system(OS) manages resources within a single server, the placement and function of the OS in disaggregated architectures are still subjects of debate in both industry and academia. Options include centralizing the OS at a single point [1, 35] in the architecture or disaggregating its functions across different resource nodes [2].

Performance overheads. Transitioning existing applications to a disaggregated architecture transparently introduces a spectrum of performance challenges. These include, but are not limited to, managing memory partitioning [36] and addressing applications with irregular memory access patterns [35]. Various other issues, such as latency sensitivity, bandwidth limitations, and the overhead of remote resource management, compound this complexity. These factors contribute to the overall performance penalty that disaggregated systems must carefully consider and mitigate.

Future interconnects. The use of networks as interconnects for resource disaggregation has been a focus of research in both academia and industry. However, networks face inherent challenges, such as performance slowdowns compared to intra-server resource access and the absence of built-in coherency. Emerging hardware technologies like Compute Express Link (CXL) [10, 11, 37] offer promising improvements, including faster access times and hardware-supported cache coherence. Despite this potential, current hardware prototypes and software support for these technologies are still in early stages, and their full impact on software design remains uncertain.

1.2 Thesis Overview

In this dissertation, we attempt to take a top-down approach and explore the optimal memory management solutions for three most significant layers, i.e. Service, OS and Hardware layers of disaggregated memory architectures.

1.2.1 Service Layer: Memory Management as a Service

As the layer closest to the application, we first explore the design requirements and challenges of providing disaggregated memory management as a service. This service manages a pool of memory resources and exposes them to applications. We propose an end-to-end system design called Jiffy, which enables multiple applications or tasks to efficiently share memory resources in an elastic manner. Jiffy also offers interfaces for several popular data structures, making it easily applicable to existing cloud applications.

1.2.2 OS Layer: In-Network Memory Management

While applications may use Jiffy to manage memory resources, we take a deeper approach by considering the Operating System (OS) as the manager of memory. This allows applications to run transparently while benefiting from the disaggregated architecture. In disaggregated systems, compute and memory resources are decoupled, meaning there is no single host, as in monolithic architectures, to handle the critical task of resource management—traditionally done by the OS. We propose a new generation OS design that embeds OS functionality within the interconnects. Starting with a system called MIND, we address fundamental challenges in memory management, such as memory address translation, memory protection, and cache coherence across multiple hosts. This resource decoupling and in-network memory management perform well for cache-friendly workloads but struggle with cache-unfriendly workloads due to the overhead of interconnect communication. To address this, we developed PULSE, a near-memory accelerator designed from the ground up. PULSE analyzes common pointer traversal applications and identifies a simple, yet effective interface that can be easily integrated into existing cloud applications.

1.2.3 Hardware Layer: Memory Management for Next-Gen Interconnects

In prior work [1,2], Ethernet has been the most commonly used interconnect for disaggregated data centers. However, with the emergence of new memory interconnects like Compute Express Link (CXL), memory management must be adapted to accommodate these new interfaces. In the context of disaggregated architectures, new challenges arise, such as how applications can effectively utilize multiple tiers of memory. To address this, we begin with a performance evaluation of CXL 1.1 extended memory in a single-host environment and explore how modern data center applications can benefit from such disaggregated memory systems.

1.3 Outline and Previously Published Work

This dissertation is organized as follows. Chapter 2 introduces Jiffy, a distributed memory management system that decouples memory capacity and lifetime from compute in the serverless paradigm. Chapter 3 describes two innovated system design: (1) MIND, a rack-scale memory disaggregation system that uses programmable switches to embed memory management logic in the network fabric. (2) PULSE, a framework centered on enhancing in-network optimizations for irregular memory accesses within disaggregated data centers. Chapter 4 presents our exploration in latest Compute Express Link(CXL) hardware. We conclude with our contributions and possible future work directions in Chapter 5.

Chapter 2 revises material from [36][1]. Chapter 3 revises material from [1][2] and [35][3]. Finally, Chapter 4 revises material from [38][4].

1. Work done in collaboration with Rachit Agarwal, Aditya Akella, and Ion Stoica

2. Work done in collaboration with Seung-seob Lee, Yanpeng Yu, Lin Zhong and Abhishek Bhattacharjee

3. Work done in collaboration with Seung-seob Lee and Abhishek Bhattacharjee

4. Work done in collaboration with the Bytedance Infrastructure team

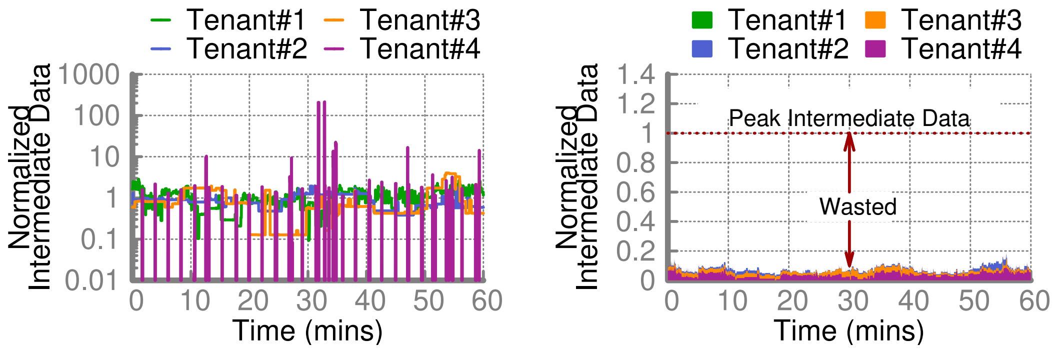
Chapter 2

Service Layer: Memory Management as a Service

We begin by focusing on the service layer, which sits above the OS layer and plays a critical role in enabling efficient memory sharing across multiple compute and memory nodes in disaggregated architectures. This layer offers greater flexibility than the OS, allowing it to provide adaptable services that cater to the specific needs of different applications. However, this flexibility comes at the cost of potential significant modifications to applications, especially when decoupling storage and compute resources is not straightforward. Without proper decoupling, developers may face substantial challenges in adapting their applications to fully utilize memory management services.

To address this, we start with data analytics applications in serverless computing [39–54], a widely adopted workload in modern data centers. Serverless architectures inherently offer on-demand elasticity, decoupling compute and storage resources logically. Recent advances in serverless analytics have demonstrated the benefits of using serverless architectures for resource- and cost-efficient data analytics. In these systems, remote, low-latency, high-throughput disaggregated memory is used to store intermediate states for inter-task[1] communication and multi-stage jobs, extending the lifetime of data beyond the task that produced it. This natural separation of compute and memory makes serverless computing an ideal candidate for leveraging disaggregated memory

1. Existing distributed programming frameworks, while different in underlying programming models and semantics, share a common structure (Figure 2.2) — the job is split into multiple *tasks*, possibly organized along multiple stages or a directed acyclic graph. Each task generates *intermediate* data during its execution; upon completion, each task partitions its intermediate data and exchanges it with tasks in the next stage.



(a) Intermediate data (normalized by mean usage) (b) Cumulative intermediate data (normalized by peak usage)

Fig. 2.1: Analysis of production workloads from Snowflake [39] for four tenants over a 1 hour window: (a) the ratio of peak to average storage usage for a job can vary by an order of magnitude during its execution; and (b) provisioning for peak usage results in average utilization $< 10\%$. Across all tenants, the average utilization is 19%.

architectures.

As discussed in [40,42,45,55], these applications handle user requests in the form of jobs, each defining its memory needs upon creation. The dilemma of balancing performance with resource efficiency for job-level memory allocation has been extensively studied [56,57]. If a job is based on average demand, performance may decline during peak demand periods due to inadequate memory, causing data spillage to slower secondary storage, such as SSDs. Conversely, allocating memory for peak demands leads to underutilization of resources when the actual demand is below peak. Evaluations on Snowflake's workload, as shown in [56], indicate a significant fluctuation in the ratio of peak to average demands, sometimes varying by two orders of magnitude within minutes.

Designing a memory management service for such systems is a non-trivial task. We begin by outlining the essential requirements for memory management in disaggregated environments, focusing on the unique challenges posed by disaggregation. We then discuss our efforts to address these challenges and suggest potential future directions for research in this rapidly evolving field.

Elasticity. Memory usage in modern computing is highly variable, with applications facing fluctuating demands [36]. Elasticity enables dynamic memory allocation based on current needs, optimizing resource utilization. Applications like data analytics consist of jobs with multiple tasks that communicate via intermediate memory. Traditional solutions allocate memory at the job level, where jobs specify their requirements before execution, and the system reserves that amount for the job's duration [42]. This approach creates a tradeoff: allocating for average demand risks performance degradation due to swapping data to slower storage (e.g., S3), as shown in Figure 2.1(a), while allocating for peak demand leads to resource waste (Figure 2.1(b)). Recent studies report that intermediate data sizes can vary by orders of magnitude during a job's lifetime [39]. For example, Figure 2.1 shows that in a Snowflake dataset with over 2000 tenants, peak-to-average memory demand can vary by two orders of magnitude within minutes, resulting in performance degradation and resource inefficiency in job-level allocations.

Isolation. The second requirement is the isolation between different compute tasks. Since multiple computing threads can be using the same disaggregated memory pool, it's essential to multiplex between applications to improve resource efficiency but at the same time keep the memory of different threads isolated from each other, which means that the memory usage of a particular application should not affect other existing applications. The number of tasks reading and writing to the shared disaggregated memory can change rapidly in serverless analytics which makes the problem even more severe.

Lifetime management. Decoupling compute tasks from their intermediate storage means that the tasks can fail independent of the intermediate data, therefore we need mechanisms for explicity lifetime management of intermediate data.

Data repartitioning. Decoupling tasks from their intermediate data also means that data partitioning upon elastic scaling of memory capacity becomes challenging, especially for certain data types used in serverless analytics (e.g. key-value store). If it's the application's responsibility to perform such repartitioning, it will involve large network transfers betweem compute tasks and the far memory system and massive read/write operations every time the capacity is scaled. What's more, the application need to implement different partitioning strategies for different kind of data structures used. Therefore, new mechansims to efficiently enable data partitioning within the far memory system is essential.

We present Jiffy, an elastic disaggregated-memory system for stateful serverless analytics. Jiffy allocates memory resources at the granularity of small fixed-size memory blocks - multiple memory blocks store intermediate data for individual tasks within a job. Jiffy design is motivated by virtual memory design in operating systems that also does memory allocation to individual process at the granularity of fixed-size memory blocks(pages). Jiffy adapts this design to stateful serverless

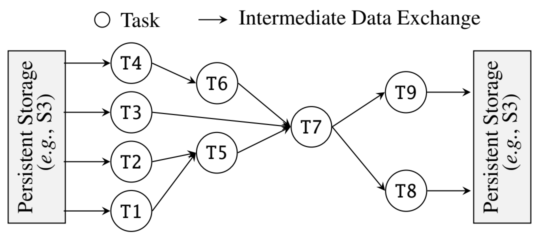


Fig. 2.2: **Execution DAG example for a typical analytics job.** Intermediate data exchange across tasks occurs via Jiffy.

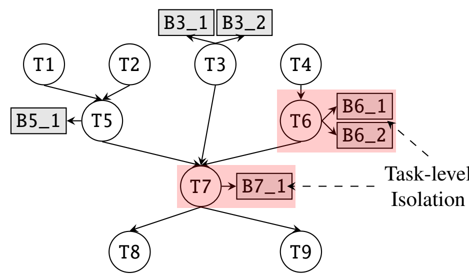


Fig. 2.3: **Hierarchical addressing** for the job in Figure 2.2. Jiffy provides task-level resource isolation for ephemeral storage under each task address-prefix (§2.1.1). Note that block addresses are only assigned to address-prefixes with currently allocated blocks; for tasks `T1`, `T2` and `T4`, blocks are directly read from persistent storage and not stored in Jiffy.

analytics. Performing resource allocation at the granularity of small memory blocks allows Jiffy to elastically scale memory resources allocated to individual jobs without a priori knowledge of intermediate data sizes and to meet the instantaneous job demands at seconds timescales. As a result, Jiffy can efficiently multiplex the available faster memory capacity across concurrently running jobs, thus minimizing the overheads of reads and writes to significantly slower secondary storage (e.g., S3 or disaggregated storage)

2.1 Jiffy Design

This section explains how Jiffy uses hierarchical addressing, intermediate data lifetime management, and flexible data repartitioning to meet these requirements. We illustrate this with Figure 2.2, which depicts the execution plan of a typical analytics job. The plan is represented as a directed acyclic graph (DAG), where nodes are computation tasks (implemented as serverless functions[2]), and edges represent intermediate data exchanged via Jiffy.

2. Functions refer to basic computation units in serverless architectures, such as Amazon Lambdas [58], Google Functions [59], and Azure Functions [60]

2.1.1 Hierarchical Addressing

Analytics jobs are often structured as multiple stages or a directed acyclic graph (DAG). In serverless analytics, where compute elasticity is key, each job can run tens to thousands of tasks [39–53]. Fine-grained resource allocation requires efficient mapping between tasks and their storage blocks, especially with rapidly changing task concurrency. High concurrency demands task-level isolation, ensuring that task arrival or departure doesn't affect others, avoiding performance degradation.

Jiffy adopts a hierarchical addressing mechanism, inspired by the Internet's IP addressing, to maintain task-to-storage mappings and ensure task-level isolation. Jiffy organizes intermediate data in a virtual address hierarchy based on task dependencies in the DAG. Internal nodes represent tasks, and leaf nodes represent Jiffy blocks storing data. Block addresses are defined by the hierarchy path, with task-generated prefixes. Dependencies between tasks are captured by edges between nodes. Jiffy builds this hierarchy from execution plans (e.g., AWS Step Functions, Azure Durable Functions) or dynamically deduces it via the Jiffy API, supporting dynamic query plans without predefined DAGs.

Example. Figure 2.3 illustrates the address hierarchy for the job in Figure 2.2. Internal nodes `T1`-`T9` represent tasks in the DAG, while leaf nodes `B3_1`, `B3_2`, etc., represent data blocks allocated by Jiffy for intermediate data storage. Edges like (`T1`, `T5`) and (`T2`, `T5`) indicate that `T5` depends on the intermediate data from both `T1` and `T2`. The full address of block `B6_2` under `T6` is `T4.T6.B6_2`, with `T4.T6` identifying all blocks under `T6`. Jiffy constructs the address hierarchy either using the execution plan from Figure 2.2 or deduces it dynamically. For instance, Jiffy can infer that since `T7`'s sub-tasks access data from `T3`, `T5`, and `T6`, these tasks must be its parents in the hierarchy.

By organizing intermediate data in a hierarchy, Jiffy manages resource allocation per address prefix. If one prefix spills to persistent storage (via Pocket), it doesn't affect others. Blocks remain assigned until reclaimed or leases expire (§2.1.2), ensuring task-level isolation regardless of churn. Like virtual memory isolating processes, Jiffy uses hierarchical addressing to isolate tasks based on the job's structure.

Two design considerations arise: (1) Jiffy's fine-grained allocation is independent of fairness policies, which can be layered on top, and (2) address translation from virtual to physical storage happens at a centralized metadata server (like Pocket [42]), scaling to arbitrary DAG sizes. Despite

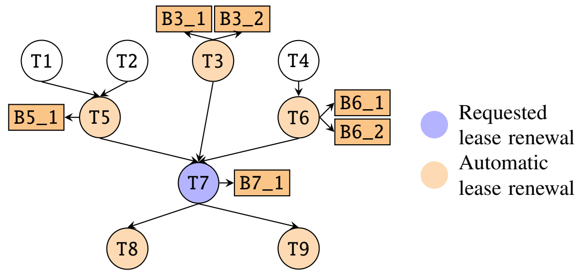


Fig. 2.4: **Lease Renewal via Address Hierarchy.** Hierarchical addressing simplifies lease renewal in Jiffy (§2.1.2), since lease renewal for an address-prefix automatically implies renewals for all parent and descendent address-prefixes in the hierarchy.

added complexity, Jiffy scales to ∼45K requests/sec/core, sufficient for most deployments.

Block sizing. Jiffy balances metadata storage and memory use with block sizes, like 128MB in HDFS [61]. Larger blocks reduce metadata but risk fragmentation, while smaller blocks improve utilization at a metadata cost. Jiffy mitigates this via fine-grained access and data repartitioning.

Isolation granularity. Task-level isolation, where nodes in the hierarchy map to tasks, is default, but finer or coarser isolation (e.g., table-level or stage-level) can be configured via the Jiffy API.

2.1.2 Data Lifetime Management

Existing ephemeral storage systems manage data at the job level, reclaiming storage when the job deregisters. In serverless analytics, decoupled task execution and storage can lead to orphaned data. Jiffy addresses this by integrating lease management [62–64] with hierarchical addressing for task-level data management. Each address prefix has a lease, and data is retained as long as the lease is renewed. Serverless platforms can trigger lease renewals during task monitoring.

Using the DAG hierarchy, Jiffy renews leases for dependent and ancestor tasks automatically, reducing overhead while preventing orphaned data. This strikes a balance between age-based eviction and explicit resource management, ensuring efficient reassignment of resources upon task or job failure.

Example. In Figure 2.2, task `T7`'s job periodically renews the lease for the prefix `T4.T6.T7`[3]. Renewing `T7`'s lease also renews those for parent tasks (`T3`, `T5`, `T6`) and descendants (`T8`, `T9`), as shown in Figure 2.4. This ensures that both parent and descendant tasks' data remain accessible.

3. Task `T7` has four address prefixes; the job can renew any.

Lease duration. Lease duration trades off control plane bandwidth with system utilization. Longer leases reduce network traffic but may delay resource reclamation. Configuring lease durations, as studied in prior work [62, 63], allows Jiffy to meet specific deployment goals.

2.1.3 Flexible Data Repartitioning

Decoupling compute tasks from their intermediate data in serverless analytics introduces challenges in achieving fine-grained elasticity for ephemeral storage. Specifically, when storage is allocated or deallocated for a task, the intermediate data must be efficiently repartitioned across available blocks. However, due to the decoupling of compute from storage and the large number of concurrent tasks, this repartitioning should not be managed by the application itself. For instance, many serverless analytics systems [41, 42] rely on key-value stores for intermediate data. If compute tasks were responsible for repartitioning during memory scaling, they would need to read key-value pairs over the network, compute new partitions based on updated memory, and write the data back to the store—resulting in significant network latency and bandwidth overhead.

Jiffy supports various data structures commonly used in data analytics frameworks, including files [39,44,50–52], key-value pairs [40–42,45,47,49,53], and queues [43,46]. Analytics jobs using these structures can delegate intermediate data repartitioning to Jiffy during resource allocation or deallocation. Each block in a Jiffy data structure tracks its own memory usage. When usage exceeds a predefined threshold, Jiffy allocates a new block to the corresponding address-prefix[4]. The overloaded block triggers a data-specific repartitioning process, moving some of its data to the newly allocated block. Conversely, when block usage drops below a low threshold, Jiffy merges it with another low-usage block before deallocating the unused block. By allowing the block itself, rather than the compute task, to handle repartitioning, Jiffy minimizes network and compute overhead for the task. Repartitioning is done asynchronously, allowing data access to continue with minimal impact on performance.

Jiffy's supported data structures enable the serverless execution of powerful distributed frameworks like MapReduce [66, 67], Dryad [68], StreamScope [69], and Piccolo [70]. Since files, queues, and key-value stores in analytics frameworks require relatively simple repartitioning (unlike

4. Similar to existing systems [24, 25, 42, 65], Jiffy can scale cluster capacity by adding or removing servers based on free blocks. Here, we focus on fine-grained elasticity.

API Group	Function Signature
	connect(honeycombAddress)
Address Hierarchy	createAddrPrefix(addr, parent, optionalArgs) createHierarchy(dag, optionalArgs) flushAddrPrefix(addr, externalPath) loadAddrPrefix(addr, externalPath)
Lease Operations	leaseDuration = getLeaseDuration(addr) renewLease(addr)
Data Structure	ds = initDataStructure(addr, type) listener = ds.subscribe(op) notif = listener.get(timeout)

Table 2.1: **Jiffy User-facing API**: Functions for connecting, managing address hierarchies, handling leases, and interacting with data structures.

complex structures such as B-trees), serverless applications can leverage Jiffy 's flexible repartitioning mechanism without requiring any modifications.

Thresholds for Elastic Scaling. In Jiffy, the high and low thresholds play a crucial role in balancing network bandwidth usage, task performance, and overall system utilization. Setting thresholds too high or too low can impact elastic scaling behavior—if scaling is triggered too infrequently, it may reduce network traffic, but it can also result in inefficient block utilization, such as numerous underutilized blocks. The optimal values for these thresholds depend heavily on the specific workload characteristics, as highlighted in previous studies [71,72]. To accommodate diverse workloads, Jiffy makes these thresholds fully configurable, allowing users to adjust them to suit their performance and efficiency needs.

2.2 Jiffy Implementation

Jiffy builds on Pocket, inheriting its scalable and fault-tolerant metadata plane, multi-tiered data storage, system-wide capacity scaling, and analytics execution model. However, Jiffy introduces hierarchical addressing, lease management, and efficient data repartitioning to address the unique challenges of serverless environments. Below, we describe the Jiffy interface and implementation, highlighting these key features.

2.2.1 Jiffy Interface

We describe the Jiffy interface in terms of its user-facing API (Table 2.1) and internal API (Figure 2.5).

User-facing API. The user-facing interface (Table 2.1) is centered around two core abstractions: *hierarchical addresses* and *data structures*. Jobs can create a new address-prefix using `createAddrPrefix`, specifying the parent prefix along with optional parameters such as initial capacity. The `createHierarchy` function generates a complete hierarchy from an execution plan (DAG), while `flush` and `load` facilitate persisting and retrieving address-prefix data from external storage (e.g., S3). Three built-in data structures can be initialized for an address-prefix via `initDataStructure`, and new structures can be defined using the internal API.

Similar to existing systems [24, 73], Jiffy 's data structures provide a notification interface, allowing tasks that consume intermediate data to be informed when new data is available. For example, a task can `subscribe` to write operations on its parent task's data structure and receive a `listener` handle. When data is written, Jiffy asynchronously notifies the `listener`, which the task can access via `listener.get()`.

```
block = ds.getBlock(op, args) // Get block
block.writeOp(args) // Perform write
data = block.readOp(args) // Perform read
block.deleteOp(args) // Perform delete
```

Fig. 2.5: **Jiffy Internal API.** The block interface is used internally in Jiffy to implement the data structure APIs.

Internal API. The data layout within Jiffy blocks is tailored to the specific data structure that owns it. Therefore, Jiffy blocks expose a set of data structure-specific *operators* (Figure 2.5) that define how requests are *routed* across blocks and how data is *accessed* or *modified*. These operators are used internally by Jiffy for its built-in data structures (§2.2.3) and are not directly exposed to jobs.

The `getBlock` operator determines the target block for an operation based on the operation type and its arguments (e.g., key hashes for a KV-store), returning a handle to the appropriate block. Each Jiffy block provides `writeOp`, `readOp`, and `deleteOp` operators, which implement data structure-specific access logic (e.g., `get`, `put`, and `delete` in a KV-store). Jiffy executes these operators *atomically* using sequence numbers but does not support atomic transactions across multiple operators.

Table 2.2: **Jiffy Data Structure Implementations**. See §**??** for details.

	Data Structure	writeOp	readOp	deleteOp	getBlock	repartition
Built-in	File (§2.2.3)	`write`	`read`	-	File offsets.	✗
	FIFO Queue (§2.2.3)	`enqueue`	`dequeue`		Tail/head.	✗
	KV-Store (§2.2.3)	`put`	`get`	`delete`	Key hash.	✓
Custom data structures						

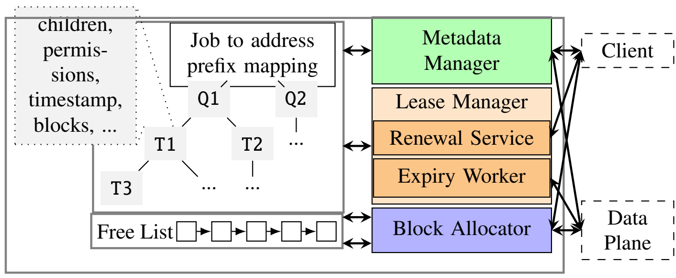


Fig. 2.6: **Jiffy controller.** See §2.2.2 for details.

2.2.2 System Implementation

Since Jiffy design builds on Pocket, its high-level design components are also similar, except for one difference: Jiffy combines the control and metadata planes into a unified control plane. We found this design choice allowed us to significantly simplify interactions between the control and metadata components, without affecting their performance. While this does couple their fault-domains, standard fault-tolerance mechanisms (*e.g.*, the one outlined in [42]) are still applicable to the unified control plane.

Control plane. The Jiffy controller (Figure 2.6) maintains two pieces of system-wide state. First, it stores a *free block list*, which lists the set of blocks that have not been allocated to any job yet, along with their corresponding physical server addresses. Second, it stores an address hierarchy per-job, where each node in the hierarchy stores variety of metadata for its address-prefix, including access permissions (for enforcing access control), timestamps (for lease renewal), a block-map (to locate the blocks associated with the address-prefix in the data plane), along with metadata to identify the data structure associated with the address-prefix and how data is partitioned across its blocks. The mapping between jobIDs (which uniquely identify jobs) and their address hierarchies is stored in a hash-table at the controller.

While the block allocator and metadata manager are similar to their counterparts in Pocket, the lease manager implements lifetime management in Jiffy. It comprises a lease renewal service that

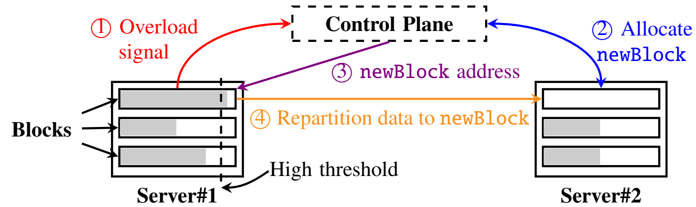


Fig. 2.7: **Data repartitioning on scaling up capacity.** Scaling down capacity employs a similar approach (§2.2.2).

listens for renewal requests from jobs and updates the lease renewal timestamp of relevant nodes in its address hierarchy, and a lease expiry worker that periodically traverses all address hierarchies, marking nodes with timestamps older than the associated lease period as expired. Finally, Jiffy adopts mechanisms from Pocket to facilitate control plane scaling and fault tolerance; we refer the reader to [42] for details.

Data plane. Jiffy data plane is responsible for two main tasks: providing jobs with efficient, data-structure specific atomic access to data, and repartitioning data across blocks allocated by the control plane during resource scaling. It partitions the resources in a pool of storage servers across fixed sized blocks. Each storage server maintains, for the blocks managed by it, a mapping from unique blockIDs to pointers to raw storage allocated to the blocks, along with two additional metadata: data structure-specific operator implementations as described in §**??**, and a subscription map that maps data structure operations to client handles that have subscribed to receive notifications for that operation.

Data repartitioning for a Jiffy data structure is implemented as follows: when a block's usage grows above the high threshold, the block sends a signal to the control plane, which, in turn, allocates a new block to the address-prefix and responds to the overloaded block with its location. The overloaded block then repartitions and moves part of its data to the new block (see Figure 2.7); a similar mechanism is used when the block's usage falls below the low threshold.

For applications that require fault tolerance and persistence for their intermediate data, Jiffy supports chain replication [74] at block granularity, and synchronously persisting data to external stores (*e.g.*, S3) at address-prefix granularity.

2.2.3 Programming Models on Jiffy

We now describe how Jiffy's built-in data structures (Table 2.2) enable various distributed programming frameworks on serverless platforms (§2.2.3-§2.2.3).

Map-Reduce Model

A Map-Reduce (MR) program [66] consists of map functions that process input key-value (KV) pairs to generate intermediate KV pairs, and reduce functions that merge all intermediate values for the same intermediate key. MR frameworks [66, 67, 75] parallelize map and reduce functions across multiple workers. Data exchange between map and reduce workers occurs via a shuffle phase, where intermediate KV pairs are distributed to ensure that values with the same key are routed to the same worker.

In Jiffy, MR executes map/reduce tasks as serverless tasks. A master process launches, tracks, and manages task failures across MR jobs. Jiffy stores intermediate KV pairs in multiple shuffle files, each containing a partitioned subset of KV pairs from all map tasks. Since multiple map tasks may write to the same shuffle file, Jiffy's strong consistency semantics ensure correctness. The master process also handles explicit lease renewals. We now describe Jiffy files in more detail.

Jiffy Files. A Jiffy file consists of multiple blocks, each storing a fixed-sized chunk of the file. The controller manages the mapping between blocks and file offset ranges at the metadata manager, and clients cache this mapping when accessing the file. The mapping is updated whenever the number of blocks allocated to the file scales. The `getBlock` operator forwards requests to the correct file block based on the request's offset range. Files support sequential `reads` and append-only `writes`. For random access, files support `seek` with arbitrary offsets, using the offset to locate the corresponding block. Since files are append-only, blocks are only added and do not require repartitioning when new blocks are added.

Dataflow and Streaming Dataflow Models

In the dataflow model, applications describe their communication patterns using directed acyclic graphs (DAGs), where DAG vertices represent computations, and data channels form directed edges between them. We refer to Dryad [68] as a reference dataflow execution engine, where channels can be files, shared memory FIFO queues, etc. The Dryad runtime schedules DAG vertices based on their dataflow dependencies: a vertex is scheduled once all its input channels are ready. A file channel is ready if its data has been fully written, while a queue is ready if it contains any data. Streaming dataflow [69] adopts a similar approach but operates on continuous event streams.

On Jiffy, each DAG vertex corresponds to a serverless task, with a master process managing vertex scheduling, fault tolerance, and lease renewals. Jiffy uses FIFO queues and files as data channels. Queue-based channels are considered ready as long as a vertex is writing to them, and Jiffy allows downstream tasks to efficiently detect item availability via notifications, described below.

Jiffy Queues. The FIFO queue in Jiffy is implemented as a growing linked-list of blocks, each storing multiple data items and a pointer to the next block. Queue size can be limited by setting a `maxQueueLength`. The controller manages only the head and tail blocks of the queue, and clients cache and update this information when blocks are added or removed. The FIFO queue supports `enqueue` and `dequeue` operations for adding and removing items. The `getBlock` operator routes these operations to the current tail and head blocks, respectively. Unlike other structures, queues do not require repartitioning. The FIFO queue uses Jiffy 's notification system to asynchronously detect when there is space to add items or when data is available for consumption through subscriptions to `enqueue` and `dequeue` events.

Piccolo

Piccolo [70] is a data-centric programming model that allows distributed machines to share mutable state. Piccolo kernel functions define sequential application logic, while sharing state with concurrent kernel functions via a KV interface. Centralized control functions create and coordinate both shared KV stores and kernel instances. Concurrent updates to the same key are resolved using user-defined accumulators.

On Jiffy, Piccolo kernel functions execute across serverless tasks, while control tasks run on a centralized master. Shared state is stored in Jiffy 's KV-store data structures (described below), which may be created per kernel function or shared across multiple functions as needed. The master periodically renews leases for Jiffy KV-stores, and, like Piccolo, Jiffy checkpoints KV-stores by flushing them to external storage.

Jiffy KV-store. The Jiffy KV-store hashes each key into one of H hash slots, where $H = 1024$ by default. The KV-store shards KV pairs across multiple Jiffy blocks, with each block responsible for one or more hash slots. A hash slot is fully contained within a single block. The controller manages the mapping between blocks and their corresponding hash slots, and this mapping is cached at the client and updated during scaling. Each block stores KV pairs as a hash table. The KV-store supports standard `get`, `put`, and `delete` operations via `readOp`, `writeOp`, and `deleteOp` operators. The `getBlock` operator routes requests to blocks based on key hashes.

Unlike files and queues, the KV-store requires repartitioning when blocks are added or removed. When a block becomes nearly full, Jiffy splits half of its hash slots into a new block, moves the relevant KV pairs, and updates the mapping at the controller. Similarly, when a block is underutilized, its hash slots are merged with another block.

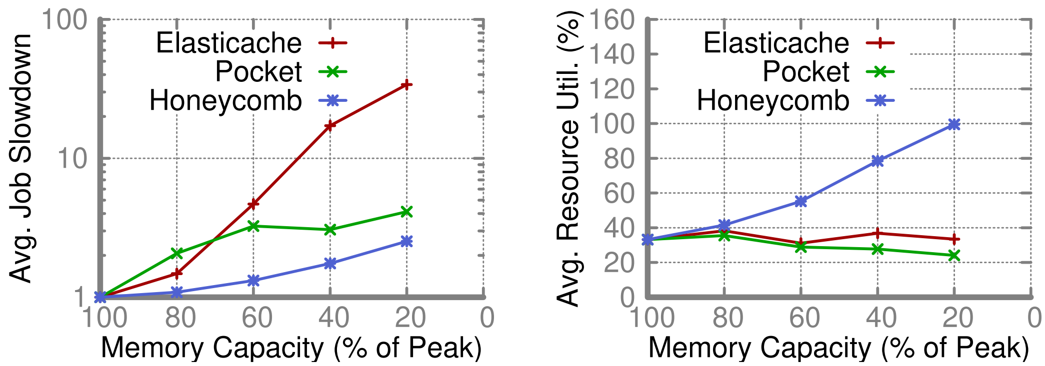
2.3 Evaluation

Jiffy is implemented in 25K lines of C++, with client libraries in C++, Python, and Java (each around 1K LOC), in addition to the original Pocket codebase. In this section, we evaluate Jiffy to showcase its benefits (§2.3.1, §2.3.2) and analyze the contributions of individual Jiffy mechanisms to overall performance (§2.3.1). Lastly, we assess Jiffy 's controller overheads in §A.1.1.

Experimental setup. Unless specified otherwise, each intermediate storage system in our experiments is deployed across 10 m4.16xlarge EC2 [76] instances, while serverless applications are hosted on AWS Lambda [76]. Since Jiffy builds on Pocket's design, it supports adding new instances to increase overall system capacity. However, our experiments do not evaluate the overheads of scaling system capacity, as this is orthogonal to Jiffy 's focus. Instead, we concentrate on multiplexing the available storage capacity for higher utilization, reducing the need to add more resources. Jiffy employs 128MB blocks, a 1-second lease duration, and thresholds of 5% (low) and 95% (high) for data repartitioning.

2.3.1 Benefits of Jiffy

Jiffy enables fine-grained resource allocation for serverless analytics. We demonstrate its impact on job performance and resource utilization across approximately 50,000 jobs from 100 randomly



(a) Job performance

(b) Resource utilization

Fig. 2.8: **Fine-grained task-level elasticity in Jiffy** enables (a) better job performance, and (b) higher resource utilization under constrained capacity. In (a), the slowdown is computed relative to the job completion time with 100% capacity (for this data point, Elasticache performance was 30% worse than Pocket, and Pocket performance was 5% worse than Jiffy). See §2.3.1 for details.

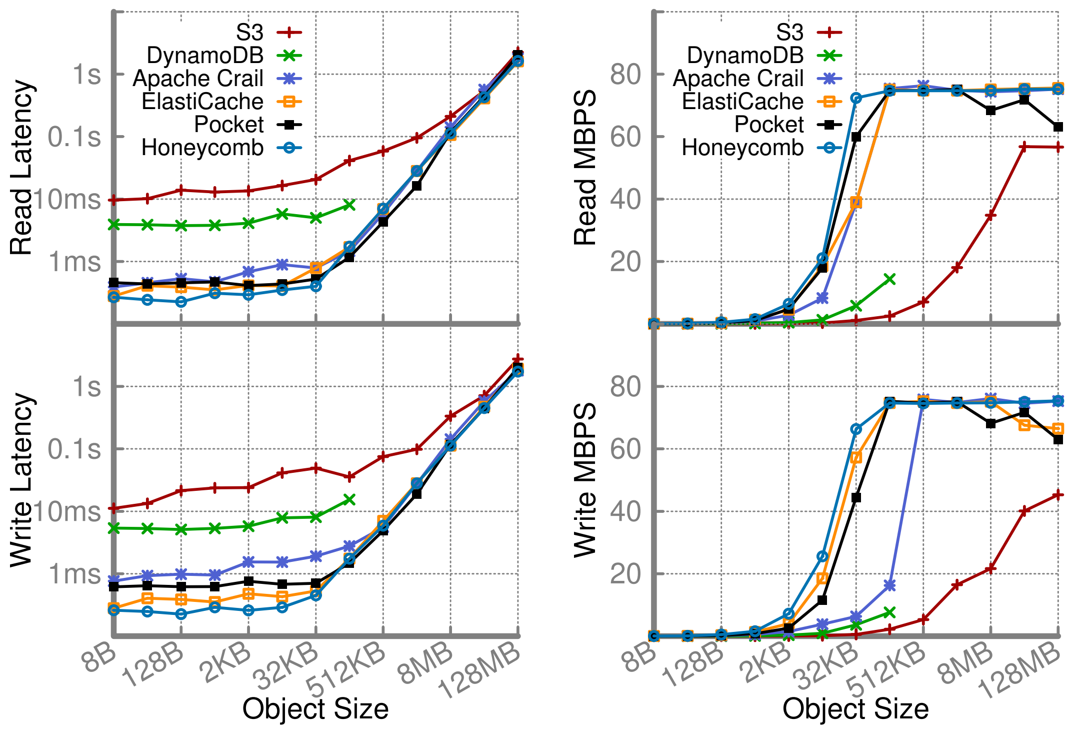
selected tenants over a 5-hour window in the Snowflake workload[5] [39].

We compare Jiffy (using the MR programming model, §2.2.3) with Elasticache [65] and Pocket [42]. Elasticache provisions resources for *all* jobs, and if capacity is insufficient, jobs must spill data to external stores like S3 [77]. Pocket, however, reserves and reclaims resources at a *job* granularity, spilling data to SSD if DRAM capacity is insufficient. Pocket's utilization can be lower than Elasticache, as it provisions separately for the peak demand of each job, which sacrifices overall utilization. To ensure a fair comparison, Pocket's control and metadata services are colocated on the same server, similar to Jiffy 's unified control plane.

Impact of fine-grained elasticity on job performance. We examine job performance under constrained intermediate storage capacity in the Snowflake workload. Figure 2.8(a) shows the average job slowdown as capacity is reduced to a fraction of peak utilization. With Elasticache, performance drops sharply when data exceeds capacity, leading to a $34\times$ slowdown at 20% capacity due to reliance on S3. Pocket experiences a $> 4.1\times$ slowdown at 20% capacity as it spills data to SSD. In contrast, Jiffy 's task-level elasticity and lease-based storage reclamation reduce data spilling, resulting in a much lower slowdown ($< 2.5\times$ at 20% capacity). This is because Jiffy multiplexes capacity more efficiently across jobs, minimizing reliance on slower storage tiers.

Impact of fine-grained elasticity on resource utilization. Figure 2.8(b) shows resource utilization under constrained capacity. While Elasticache and Pocket see reduced or stagnant utilization

5. We did not evaluate the full 14-day window with > 2000 tenants due to intractable cost overheads.



(a) Latency (b) MBPS

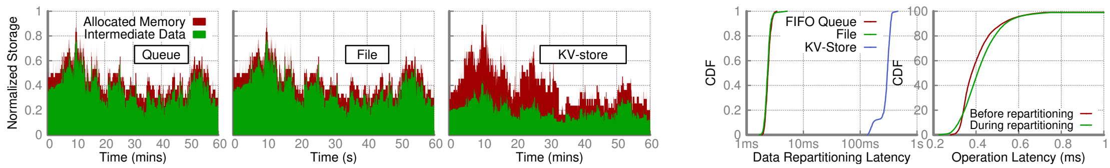
Fig. 2.9: **Jiffy performance comparison with existing storage systems (§2.3.2).** Despite providing the additional benefits demonstrated in §2.3.1, Jiffy performs as well as or outperforms state-of-the-art storage systems for serverless analytics.

as capacity is constrained, Jiffy 's utilization *improves*. Elasticache and Pocket allocate capacity at a job or coarser granularity, wasting unused resources regardless of total system capacity. In contrast, Jiffy 's fine-grained elasticity and lease-based reclamation allow it to multiplex capacity more effectively, reducing SSD spillover and improving performance as shown in Figure 2.8(a).

2.3.2 Performance Benchmarks for Six Systems

We now compare Jiffy 's performance (using its KV-Store data structure) against five state-of-the-art systems commonly used for intermediate data storage in serverless analytics: S3, DynamoDB, Elasticache, Apache Crail, and Pocket. Since only a subset of these systems support request pipelining, we disable pipelining across all of them for consistency.

To measure latency and throughput, we profiled synchronous operations issued from an AWS Lambda instance using a single-threaded client. Figure 2.9 shows that in-memory data stores like Elasticache, Pocket, and Apache Crail achieve low latency (sub-millisecond) and high throughput.



(a) Efficient lifetime management (b) Efficient data repartitioning

Fig. 2.10: **Jiffy data lifetime-management and data repartitioning.** (a) Jiffy provides fine-grained elasticity through lease-based lifetime management for its built-in data structures: FIFO Queue (left), File (center), and KV-store (right). It efficiently reclaims resources from tasks once their leases expire. (b) Jiffy enables efficient data repartitioning as allocations scale up, with repartitioning for a single block completing within 2-500ms (left). Additionally, the latency for 100KB `get` operations is minimally affected during KV-store repartitioning. Note: plots in (a) and (b) share a common y-axis; the x-axis for (c, left) is in log scale.

In contrast, persistent data stores like S3 and DynamoDB exhibit significantly higher latencies and lower throughput; note that DynamoDB only supports objects up to 128KB. Jiffy matches the performance of these in-memory data stores while also providing the additional benefits discussed in §2.3.1.

2.3.3 Understanding Jiffy Benefits

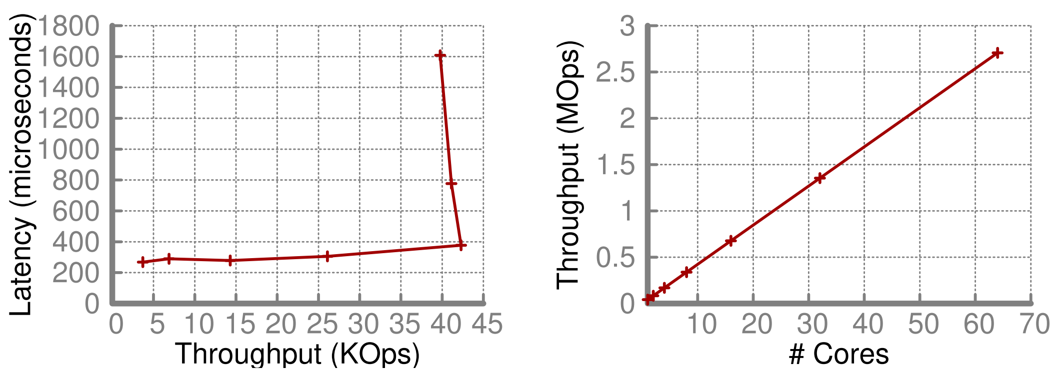
Figure 2.8 demonstrates how Jiffy's fine-grained elasticity provides performance and resource utilization advantages over other state-of-the-art systems. This elasticity is achieved through hierarchical virtual addressing, flexible data lifetime management, and data repartitioning. In this section, we isolate and evaluate the impact of these mechanisms.

Fine-grained elasticity via data lifetime management. Unlike traditional storage systems, Jiffy 's lease-based data lifetime management enables the reclamation of unused resources, reallocating them to jobs in need. Coupled with fine-grained resource allocations and efficient data repartitioning, this enables elasticity for serverless jobs. To evaluate this, we examine storage allocation across various Jiffy data structures (Figure 2.10(a)) using the Snowflake workload from Figure 2.1.

FIFO queue and file data structures exhibit seamless elasticity as intermediate data is written to them, as they do not require repartitioning. The allocated capacity slightly exceeds the intermediate data size, accounting for block metadata (e.g., object metadata for FIFO queue items) and unused space in head/tail blocks. For the KV-store, inserted keys are sampled from a Zipf distribution since the Snowflake dataset lacks access patterns. Due to the skew, some Jiffy blocks receive most key-value pairs and frequently split when their capacity grows too high, leading to higher allocated capacity. However, Jiffy 's lease mechanism quickly reclaims resources after their utility ends, ensuring that overheads are temporary.

Efficient elastic scaling via flexible data repartitioning. A key factor in Jiffy 's elasticity is its efficient data repartitioning. Figure 2.10(b) shows the CDF of repartitioning latency per block across the three data structures under the Snowflake workload. The latency includes the time from detecting an overloaded/underloaded block to the completion of repartitioning. Storage servers take ∼1-1.5ms to connect to the controller, with two round trips (100-200μs in EC2) to trigger block allocation/reclamation and update partitioning metadata. Unlike FIFO Queue and File, KV-Store requires data repartitioning across blocks, but since only half the block capacity (∼64MB) is moved, Jiffy completes repartitioning in a few hundred milliseconds over 10Gbps links, achieving block-level repartitioning with low latency (2-500ms).

Importantly, Jiffy does not block data structure operations during repartitioning. As shown in Figure 2.10(b), the CDF of 100KB `get` operations in the KV-Store before and during scaling remains almost identical, indicating minimal impact on operation latency during scaling.



(a) Controller throughput vs. latency on a single CPU core.

(b) Controller throughput scaling with multiple cores.

Fig. 2.11: **Jiffy controller performance.** Details in Appendix A.1.1.

2.3.4 Controller Overheads

Jiffy introduces several additional components at the controller compared to Pocket, including metadata management, lease management, and handling data repartitioning requests. As a result, its performance is expected to be lower than Pocket's metadata server. However, this is acceptable as long as Jiffy can manage the typical control plane request rates observed in real-world workloads, such as the peak of a few hundred requests per second—including lease renewals—seen in our evaluations and those in [42].

Figure A.1(a) shows the throughput-versus-latency curve for Jiffy controller operations on a single CPU core of an m4.16xlarge EC2 instance. The controller throughput saturates at around 42 KOps with a latency of 370μs. While this is lower than Pocket's throughput (90 KOps per core), it is more than sufficient to handle the control plane load of real-world workloads. Additionally, throughput scales almost linearly with the number of cores, as each core processes requests independently for distinct subsets of virtual address hierarchies (Figure A.1(b)). Finally, the control plane can scale across multiple servers by partitioning the address hierarchies.

Storage overheads. The task-level metadata storage in Honeycomb has a minimal overhead of just 64 bytes of fixed metadata per task and 8 bytes per block. For Jiffy's default 128MB blocks, this results in an insignificant storage overhead ($< 0.00005 - 0.0001\%$ of total storage).

2.4 Related Work

2.5 Summary

In this chapter, we have presented Jiffy, a memory management service designed for disaggregated memory, allocating memory in small fixed-size blocks. These blocks store intermediate data for individual tasks within a job. Inspired by virtual memory systems in operating systems, Jiffy scales memory resources elastically without prior knowledge of data sizes, adapting to job demands in real-time. This approach allows Jiffy to efficiently share fast memory across jobs, reducing reliance on slower secondary storage such as S3 or disaggregated storage.

Chapter 3

Operating System Layer: In-Network Memory Management

In the previous chapter, we explored the design of memory management for disaggregated architectures at the service layer. Specifically, we examined how serverless applications, which are inherently aware of disaggregated memory and compute resources, require explicit memory management for handling intermediate data. Integrating such applications with Jiffy is straightforward, as they can directly benefit from Jiffy's elasticity and lifetime management features. However, general-purpose applications (beyond serverless data analytics) are typically developed without any knowledge of the underlying hardware specifics, such as disaggregated resources. As a result, integrating these applications with external memory services (e.g., Jiffy) often necessitates significant code modifications to accommodate their APIs, which may present an undesirable burden for developers.

In traditional monolithic architectures, memory management is typically handled by the operating system (OS), which manages virtual and physical pages, performs memory address translation, enforces memory protection, and provides a simple interface to user applications. This abstraction hides the hardware details, thereby easing the burden of memory management for developers. If the OS is made aware of disaggregated architectures and can transparently manage memory, it would be possible to migrate existing applications to these architectures without requiring any code modifications.

The fundamental distinction between performing memory management at the OS layer versus the service layer lies in the scope and specificity of the functionality provided. The OS must offer highly general functionality that applies to all applications, while the service layer can afford to provide more specialized features tailored to specific application types (e.g., lifetime management in Jiffy). A key question that arises is where the OS should be situated within the disaggregated architecture. Unlike monolithic architectures, where the OS resides directly on each server, disaggregated architectures lack a single, centralized server. We observe that the network interconnect (e.g., Ethernet) presents a promising point for implementing OS-level memory management (3.1). However, even if memory management functionalities are implemented transparently within the OS, their performance may vary depending on the application type due to fundamental differences in how resources are organized (3.2).

In this chapter, we shift our focus to embedding memory management functionality directly within the OS. We first introduce MIND, an in-network OS design that enables transparent memory management for disaggregated resources. Following this, we present PULSE, an in-network near-memory accelerator designed to optimize performance for pointer traversal workloads.

3.1 MIND: In-Network Memory Management for Disaggregated Data Centers

Implementing memory management at the OS layer in disaggregated architectures poses three significant challenges. First, remote memory access requires low latency and high throughput, with targets of 10 μs latency and 100 Gbps bandwidth per compute blade [2,3,31,32]. Second, both compute and memory resources must scale elastically to meet varying workloads. Finally, widespread adoption of disaggregated memory necessitates support for unmodified applications, minimizing the need for developers to rewrite code.

We introduce MIND, the first memory management system designed for rack-scale disaggregated memory, addressing these challenges by embedding the memory management module (logic and metadata for memory management) directly within the network fabric and leveraging programmable network switches [78,79].

The placement of MIND's memory management *within the network fabric* is motivated by three

Table 3.1: **Parallels between memory & networking primitives.**

Virtual Memory	$\Longleftrightarrow$	**Networking**
Memory allocation		IP assignment
Address translation		IP forwarding
Memory protection		Access control
Cache invalidations		Multicast

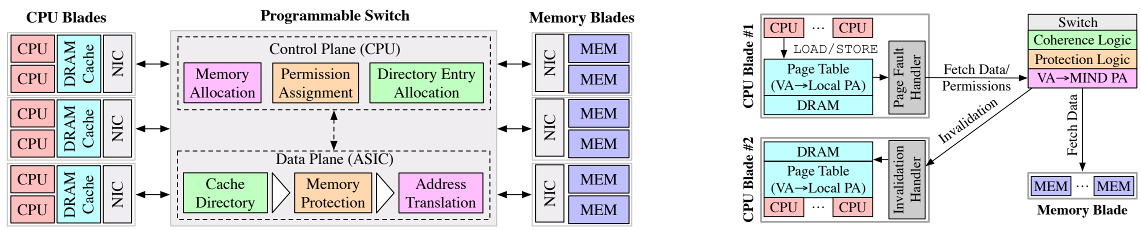


Fig. 3.1: **(left) High-level MIND architecture, and, (right) data flow for memory accesses in MIND.**

key factors: (1) its central location provides a global view of the system, enabling direct memory access without requiring metadata coherence, (2) virtual memory access bears a structural similarity to network address access 3.1, and programmable network switches [78] are capable of executing at line rate, making them suitable for implementing memory management logic, and (3) incorporating cache coherence logic into the network fabric helps reduce latency and bandwidth overhead.

MIND provides a *transparent virtual memory* abstraction to applications, functioning similarly to traditional OS mechanisms. It intercepts memory allocations on CPU blades and performs memory operations via RDMA, using a switch-based MMU for managing cache coherence. Memory blades store pages and directly handle RDMA requests, enabling true hardware disaggregation.

Figure 3.1(left) presents an overview of MIND's design, while Figure 3.1(right) illustrates its memory access flow. CPU blades run user processes and utilize local DRAM as a cache. Memory allocations and deallocations are intercepted and forwarded to the switch control plane, which manages memory allocation and access permissions. All memory operations are handled by the CPU cache, with virtual addresses translated locally. If a requested page is not cached, a page fault triggers an RDMA request to fetch it from the memory blades. Coherence updates may also trigger page faults, which are handled by the switch.

Since CPU blades do not maintain memory metadata, RDMA requests operate solely on virtual addresses. The switch's data plane intercepts these requests, managing cache coherence, permission verification, and address translation. If no cache contains the requested page, the switch forwards the request to the appropriate memory blade. MIND relies on one-sided RDMA, removing the need for CPUs on memory blades and paving the way for complete hardware disaggregation.

3.2 Need for PULSE

Recent disaggregated systems [1, 2] use small DRAM caches on CPU nodes while accessing memory across network-attached memory nodes with large DRAM pools (Fig. 3.2 (top)). Disaggregation allows flexible memory allocation across CPU and memory nodes, enabling high utilization and elasticity.

However, limited bandwidth and latency of network-attached memory remain a challenge, constrained by the speed of light. Even with near-terabit links and RDMA [80], remote memory is much slower than local memory [3]. CXL interconnects [10] show similar patterns, with 300 ns latency compared to 10–20 ns for L3 cache [37]. CPU caches can reduce average memory access latency, but their effectiveness is limited by data locality and cache size. Remote memory access is unavoidable for pointer-heavy applications, such as database index lookups [81–91] and graph analytics [92–95] (Fig. 3.3, §**??**).

Memory-intensive applications [67, 96–101] often require traversing linked structures like lists, hash tables, trees, and graphs. Despite large memory pools in disaggregated architectures, network pointer traversals remain slow [3]. Recent systems [1–3, 31, 32] mitigate this by caching hot data in CPU DRAM, but pointer traversals still suffer, as we demonstrate next.

Pointer traversals in real-world workloads. Studies [67, 94, 102–106] show that cloud applications spend 21% to 97% of their execution time on pointer traversals. We analyzed three representative cloud applications — a WebService frontend [28], WiredTiger indexing [107], and BTrDB time-series analysis [108] — using swap-based disaggregated memory [32]. Varying the CPU cache size from 6.25% to 100% of the working set, Fig. 3.3(a) shows that (i) significant execution time is spent on pointer traversals (13.6%, 63.7%, and 55.8% respectively, even with full cache), and (ii) traversal time increases as CPU cache size decreases.

Distributed traversals. As application workloads and working-set sizes grow, disaggregated systems allocate memory across multiple memory nodes [1, 2, 31, 32]. To optimize load balancing and utilization, they use fine-grained allocations (e.g., 1 GB in [2], 2 MB in [1]), but this fragments linked structures across memory nodes, leading to frequent distributed traversals.

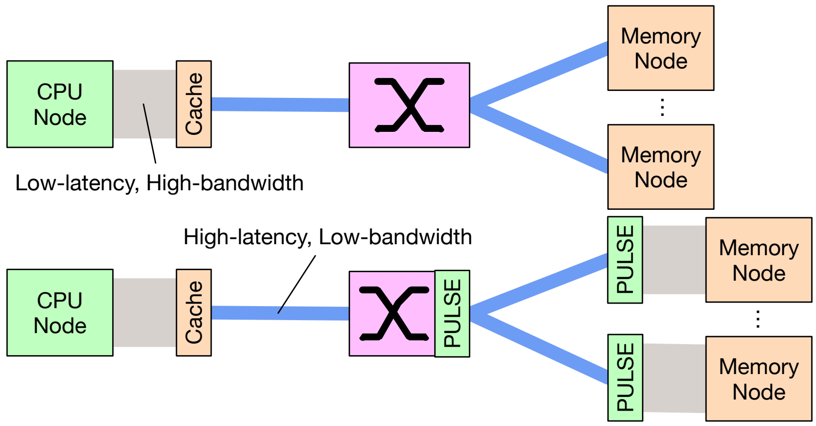
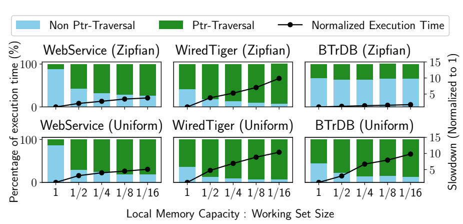
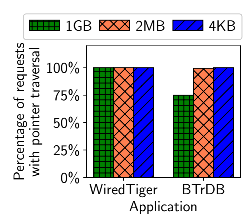


Fig. 3.2: **Need for accelerating pointer traversals.** *(top)* Pointer traversals in disaggregated architectures are limited by slow memory interconnects. *(bottom)* Like CPU caches, we propose a fast, lightweight accelerator for cache-unfriendly pointer traversals in traversal-heavy workloads.

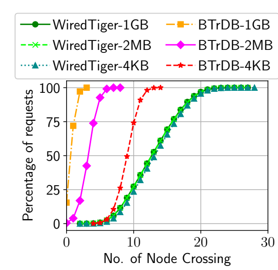
Fig. 3.3(b) illustrates this for WiredTiger and BTrDB on a setup with 1 compute and 4 memory nodes: over 97% and 75% of requests, respectively, cross memory node boundaries. Fig. 3.3(c) shows the CDF of memory node crossings. While WiredTiger's randomly ordered data requires frequent crossings, BTrDB's time-ordered data confines larger allocations to the same node, reducing crossings. However, smaller allocations, necessary for high utilization, still result in many cross-node traversals.



(a) Our empirical analysis



(b) % of distributed traversals



(c) CDF of distributed traversals

Fig. 3.3: **Time cloud applications spend in pointer traversals.**

Similar to CPU caches for quick data access, we propose adding lightweight, fast processing units to memory nodes with high-bandwidth, low-latency access for accelerating pointer traversals (Fig. 3.2 (bottom)). The interconnect should also enable efficient, scalable traversals across multiple memory nodes handling large linked data structures.

We design PULSE[1], a distributed framework for efficient pointer traversals in rack-scale disaggregated memory. PULSE addresses expressiveness, energy efficiency, and performance through a redesigned near-memory processing approach. At its core is an expressive iterator interface, providing a unified abstraction for pointer traversals in key-value stores [24, 106], databases [86–88, 90, 107], and big-data analytics [92–95] (§??). This abstraction supports a wide range of traversal-heavy workloads, enabling (i) integration with familiar toolchains and (ii) efficient hardware accelerators optimized for iterators.

PULSE ensures efficient pointer traversals via a novel accelerator that decouples logic and memory pipelines, leveraging the sequential nature of iterator execution (§??). This allows high memory utilization by balancing more memory with fewer logic pipelines. A scheduler distributes traversal logic across pipelines, using multiplexing to maximize utilization. While we implemented PULSE on FPGA-based SmartNICs due to ASIC complexity, the design targets an eventual ASIC implementation for greater efficiency.

For distributed traversals, PULSE uses a programmable network switch, treating pointer traversal across memory nodes like packet routing (§??). The switch inspects iterator requests and forwards them to the correct memory node at line rate. We implemented a prototype using commodity servers, SmartNICs, and a programmable switch, with non-invasive hardware and software changes ensuring easy deployment.

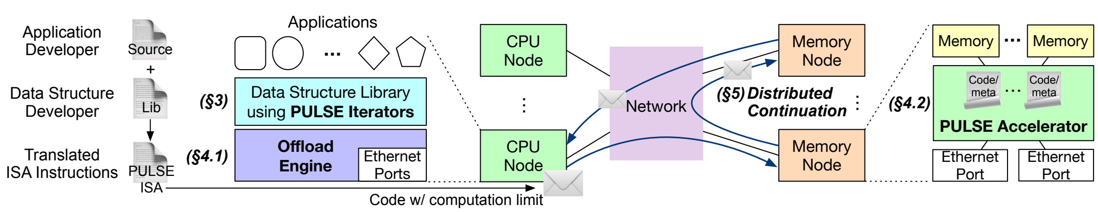


Fig. 3.4: **PULSE Overview.** Developers use PULSE's iterator interface (§??) to express pointer traversals, translated to PULSE ISA by its dispatch engine (§??). During execution, PULSE accelerator ensures energy efficiency (§3.3.3) and in-network design enable distributed traversals (§??).

1. **P**rocessing **U**nit for **L**inked **S**tructur**E**s.

3.3 PULSE Design

PULSE innovates across three key design elements (Fig. 3.4). At its core, PULSE's iterator-based programming model (§??) simplifies porting real-world data structure traversals. It supports *stateful* traversals with a *scratchpad*, allowing developers to store and update intermediate states (*e.g.*, aggregators, arrays) during iterator execution. This iterator-based approach enables tractable accelerator design and efficient distributed traversals.

The iterator code from the developer is translated into PULSE's ISA for execution by PULSE accelerators (§??). Energy efficiency and performance are achieved through a novel accelerator that uses disaggregated logic and memory pipelines, with an ISA tailored to the iterator pattern. A specialized scheduler ensures high utilization and performance in this disaggregated setup.

For scalable distributed pointer traversals, PULSE leverages programmable network switches to reroute requests crossing memory node boundaries (§??). Hierarchical address translation is performed *in-network*, with the switch handling memory node-level translation and the memory node accelerators handling local translation and protection. If a request is not local, the accelerator sends it back to the switch for rerouting.

Assumptions. PULSE relies on CPU node logic for synchronization, requiring explicit lock handling by the application. While recent efforts enable locking on NICs [29, 109] and switches [110], these are orthogonal and can be incorporated into PULSE. Finally, PULSE uses the caching scheme from prior work [28], maintaining a transparent cache in the data structure library.

3.3.1 PULSE Programming Model

We begin with PULSE's programming model, as a well-designed interface is critical for real-world traversal-heavy applications and the development of efficient pointer traversal accelerators and distributed mechanisms. PULSE's interface is intended for data structure library developers to offload pointer traversals in linked structures. Since code modifications are limited to the data structure libraries, applications using their interfaces require no changes.

We analyzed various popular data structures [111–114] and identified a common traversal pattern: (1) initializing a start pointer, (2) iteratively determining the next pointer, and (3) checking a termination condition at the end of each iteration. This pattern closely matches the *iterator* design motif, which is common across languages [113]. Thus, PULSE adopts the iterator interface as the hardware-software boundary for pointer traversals (Listing 3.1).

The interface exposes three user-implemented functions: (1) `init()` initializes the start pointer based on the data structure, (2) `next()` updates the current pointer, and (3) `end()` determines if the traversal should stop. PULSE then iteratively executes the traversal using these functions through `execute()`. We also introduce two key aspects of our iterator abstraction to enhance and constrain 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 3.1: PULSE interface.

Stateful traversals. Pointer traversals in data structures are often stateful, with varying state types. For example, hash table lookups use the search key as state, while B-Tree summations require a running total that updates with each value. To support this, PULSE iterators include a `scratch_pad` for storing arbitrary state. The state is initialized in `init()`, updated in `next()`, and finalized in `end()`. The `execute()` function returns the contents of `scratch_pad` (Line 10), allowing developers to access desired data.

Bounded computations. PULSE accelerators enable lightweight processing for memory-intensive operations, maximizing bandwidth utilization. While `init()` runs on the CPU, `next()` and `end()` are offloaded to PULSE accelerators, which impose two limitations on memory accesses and computations. First, PULSE disallows nondeterministic executions, such as unbounded loops that cannot be unrolled. Second, `execute()` in Listing 3.1 restricts the maximum iterations per request to prevent long traversals from blocking others. If the limit is exceeded, PULSE terminates the traversal and returns the `scratch_pad` value to the CPU, allowing a new request to continue from that point.

```
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 3.2: 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 3.3: PULSE realization for `unordered_map::find()`.

An illustrative example. We illustrate how the `find()` operation of the C++ STL `unordered_map` can be adapted for PULSE. Listing 3.2 presents a simplified STL implementation, where the traversal starts by computing a hash function to find the corresponding hash bucket pointer. It then iterates through the linked list in the bucket, terminating when the key is found or the list ends.

Listing 3.3 shows the iterator implementation in PULSE. The core logic remains largely unchanged, with minor adjustments to the `init()`, `next()`, and `end()` functions. The main differences lie in how the state (search key) is passed between these functions and how results are returned via the `scratch_pad` (an error message if the key is not found, or its value if it is).

3.3.2 Accelerating Pointer Traversals on a Node

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 (§**??**) 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 [115]. 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 3.2). There are, however, a few notable aspects to our adapted ISA and the translation of iterator code to it. First, as noted in §**??**, 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 [116], 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 (§3.3.3). Second, again as noted in §**??**, developers can maintain state and return values using a `scratch_pad` of pre-configured 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.

Finally, we found that the iterator traversal pattern typically can be broken down into two types of computation — fetching data[2] 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 by `cur_ptr`. For instance, the `unordered_map::find()` realization shown in Listing 3.3 makes references to `cur_ptr->key`, `cur_ptr->value`

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

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 3.2: **PULSE adapts a restricted subset of RISC-V ISA** (§**??**).

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 §3.3.3, 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 (§**??**). To recover from packet drops, the dispatch engine embeds a request identifier (ID) with the CPU node ID and a local request counter in the request packets, maintains a timer per request, and 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 [24, 106], databases [86–88, 90, 107], and big-data analytics [92–95] to PULSE's iterator interface (§??). PULSE's dispatch engine is implemented on Intel DPDK-based [117] low-latency, high-throughput UDP stack. PULSE compiler adapts the Sparc backend of LLVM [118] since its ISA is close to PULSE's ISA. Our LLVM frontend applies a set of analysis and optimization passes [119] 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.

3.3.3 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 §**??**.

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. 3.5 (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

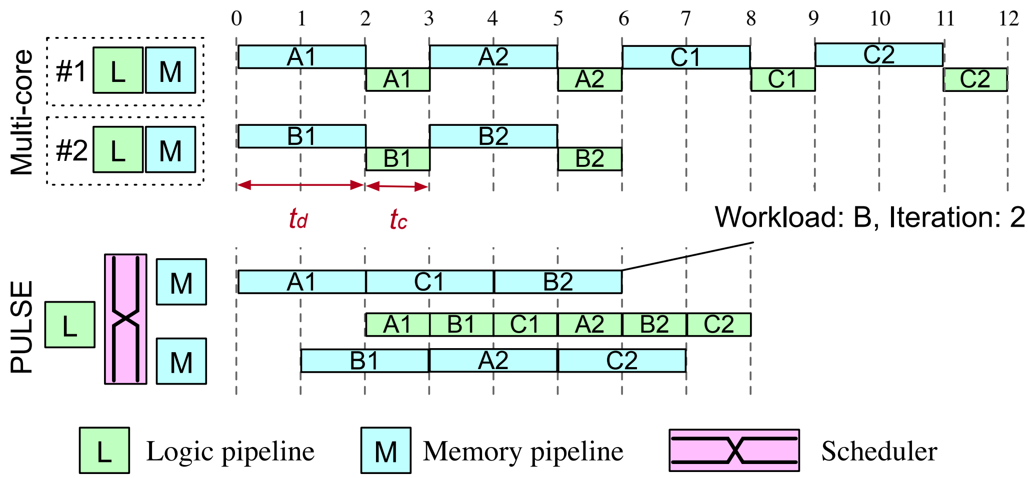


Fig. 3.5: **PULSE accelerator architecture.** (top) Traditional multi-core architectures with tightly coupled logic and memory pipelines result in low utilization and longer execution times. (bottom) PULSE accelerator's *disaggregated* design with an unequal number of logic and memory pipelines efficiently multiplexes concurrent iterator executions across them for near-optimal utilization and performance.

is busy. While thread-level parallelism permits iterator requests to 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 §**??** is $\frac{m}{n}$, i.e., there are fewer logic pipelines than memory pipelines in keeping with Property 2. Fig. 3.5 (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 (§**??**). 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

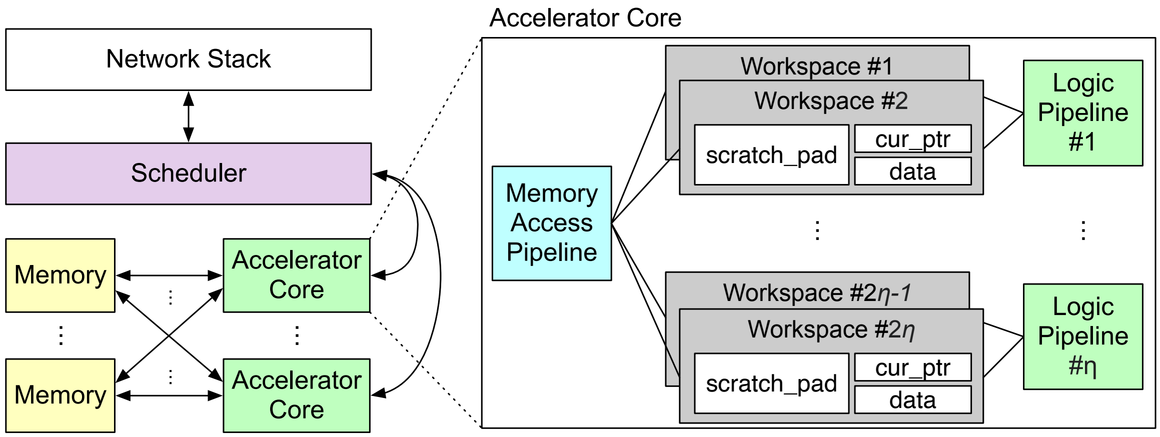


Fig. 3.6: **PULSE accelerator overview.** See §3.3.3 for details.

to fully utilize all m logic and n memory pipelines. While we omit a theoretical proof for brevity, Fig. 3.5 (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 down-scaling frequency [120]; 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 §**??**, 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.

PULSE Accelerator Components. PULSE accelerator comprises n memory and m logic pipelines for executing iterator requests, a scheduler that multiplexes requests across the logic and memory pipelines, and a network stack for parsing pointer-traversal requests from the network (Fig. 3.6).

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 [121]) 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 3.2). 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 pro-

tection 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. 3.5 (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 (§**??**).

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 (§3.4) 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 [122] (for page tables), 100 Gbps Ethernet IP, link-layer IPs [123], and burst data transfers [124] 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.

3.3.4 Distributed Pointer Traversals

By restricting pointer traversals to a single memory node (§**??**), 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

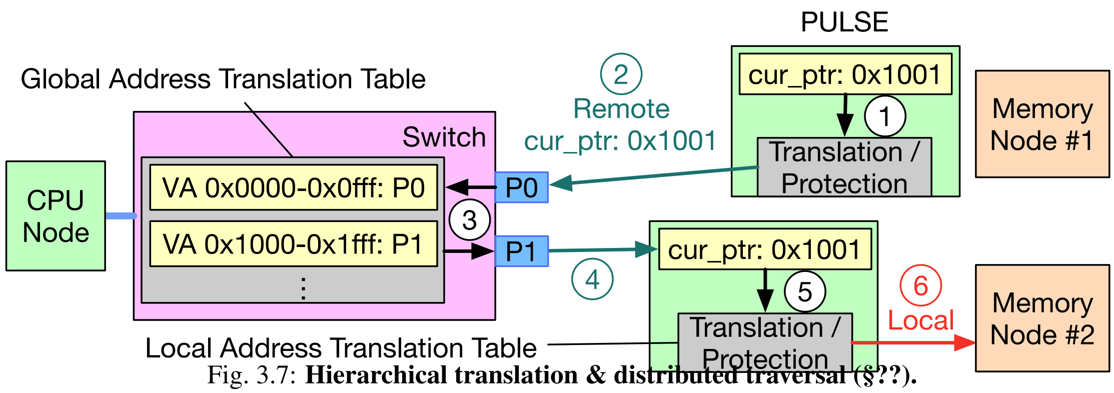


Fig. 3.7: **Hierarchical translation & distributed traversal (§??).**

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. 3.7. 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 meta-

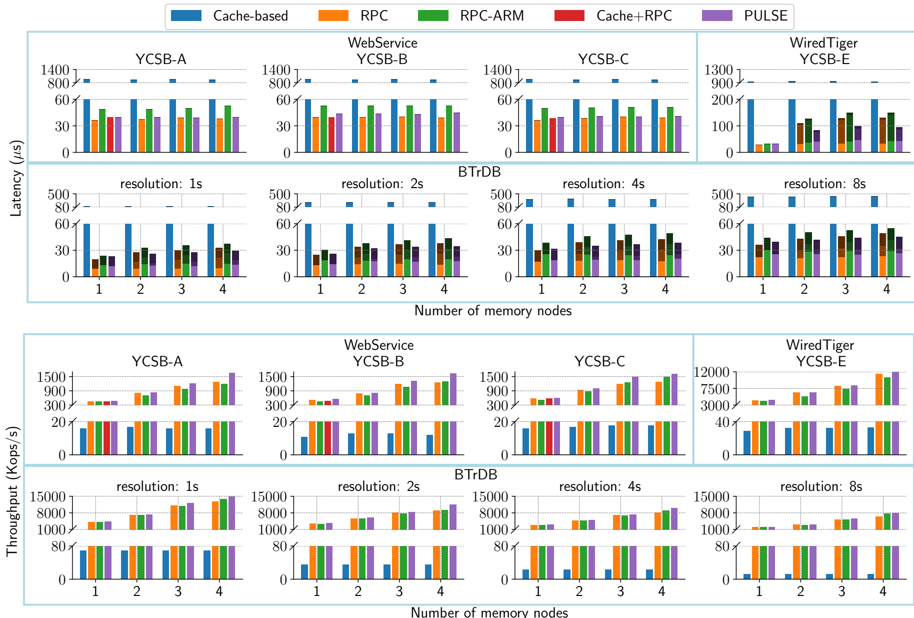


Fig. 3.8: **Application latency (top) & throughput (bottom) (§??).** The darker color indicates the time spent on cross-node pointer traversals, which increases with the number of memory nodes in WiredTiger and BTrDB.

data at the accelerator (①), as outlined in §**??**. 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 (§3.3.3), 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 confining all of the iterator state in `scratch_pad` and `cur_ptr` and 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.

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 3.3: **Workloads used in our evaluation (§3.4).** t_c and t_d correspond to compute and memory access time at the PULSE accelerator.

3.4 Evaluation

Compared systems. We compare PULSE against: (1) a **Cache-based** system that relies solely on caches at CPU nodes to speed up remote memory accesses; we use Fastswap [31] as the representative system, (2) an **RPC** system that offloads pointer-traversals to a CPU on memory nodes, (3) **RPC-ARM**, an RPC system that employs a wimpy ARM processors at memory nodes, and (4) a **Cache+RPC** approach that employs data structure-aware caches; we use AIFM [28] as the representative system. (1, 4) use a cache size of 2 GB, while (2, 3) use a DPDK-based RPC framework [125].

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 [126] 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 [127] and report energy consumption of the **minimum** number of CPU cores needed to saturate the bandwidth. We use Bluefield-2 [128] 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 3.3): (1) *Web Service* [28] that processes user requests by retrieving user IDs from an in-memory hash table, using these IDs to fetch 8KB 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 [129]. (2) *WiredTiger Storage Engine* (MongoDB backend [130]) 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 [28, 131], we model user queries using the YCSB E workload with Zipf distribution [129] on 8B keys and 240B values. (3) *BTrDB Time-series Database* [108] 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 [132]. We run stateful aggregations (sum, average, min, max) for time windows of different resolutions, from 1s to 8s, on the Open μPMU Dataset [133] with voltage, current, and phase readings from LBNL's power grid [108].

Performance for Real-world Applications

Since AIFM [28] 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. 3.8 demonstrates the advantages of accelerating pointer-traversals at disaggregated memory. Compared to the Cache-based approach, PULSE achieves 9–34.4$\times$ lower latency and 28–171$\times$ higher throughput across all applications using only one network round-trip per request. RPC-based systems observe 1–1.4$\times$ lower latency than PULSE due to their 9$\times$ 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 [28, 134] 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 8KB 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 3.3). Interestingly, the Cache-based approach performs significantly better for BTrDB than WebService and WiredTiger due to the 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. 3.8 shows that employing multiple memory nodes introduces two major changes in performance trends: (1) the latency increases when the pointer traversal spans multiple memory nodes, and (2) 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.

PULSE observes lower latency than the compared systems due to in-network support for distributed pointer-traversals (§**??**). 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$\times$ 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 [135] for PULSE (all power rails) and Intel RAPL tools [136] for RPC on CPUs [126] (CPU package and DRAM only). For RPC-ARM on ARM cores, since there is no power-related performance counter [137] or open-source tool available, we adapt the measurement approach from prior work [138]. Specifically, we calculate the CPU package's energy using application CPU cycle counts and DRAM power using Micron's estimation tool [139]. 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 [140] 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,

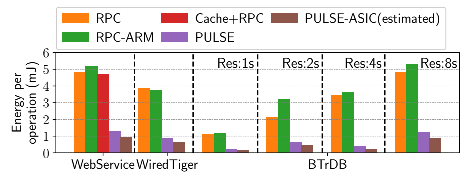


Fig. 3.9: **Application energy consumption per operation (§??).**

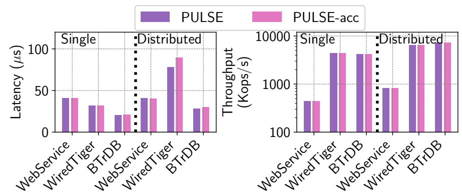


Fig. 3.10: **Impact of distributed pointer traversals (§??).**

and Cache+RPC.

Fig. 3.9 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 (§3.3.3). Our estimation shows that PULSE's ASIC realization can conservatively reduce energy use by an additional $6.3 - 7\times$ 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 [138], which attribute the increased energy use to their longer execution times, resulting in higher aggregate energy demands.

Understanding PULSE Performance

Distributed pointer traversals. We evaluate the impact of distributed pointer traversals (§**??**) 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. 3.10 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. 3.11 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,

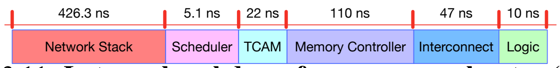


Fig. 3.11: **Latency breakdown for PULSE accelerator (§??).**

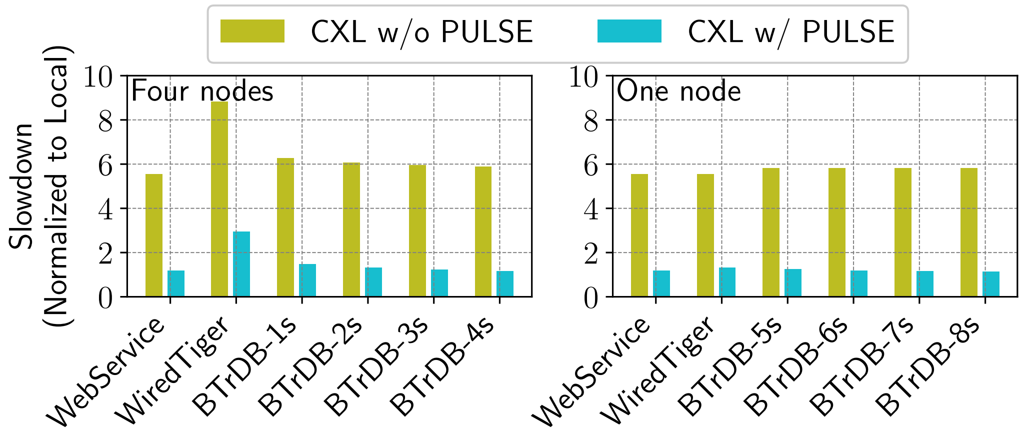


Fig. 3.12: **Slowdown with simulated CXL interconnect (§3.6).**

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.

3.5 Related Work

Prior work has explored such processing units in near-memory and processing-in-memory architectures [141–169], as well as CPUs [28, 170–174] and FPGAs [138, 175] near remote/disaggregated memory, though these approaches have notable limitations.

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 [141–169].

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 [29, 109, 176–180] 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.

Poor utilization/power-efficiency in CPUs. Many prior works have explored remote procedure call (RPC) interfaces to enable offloading computation to CPUs on memory nodes [28, 170–173]. 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 (§3.4), 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 [138, 175] or ASICs [168, 169] 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 [181], 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 [168]), 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) [182] or RDMA processing units (PUs) [183] 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) [3, 183]. 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 [138] and confirmed in our evaluation (§3.4).

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 (§**??**). Second, existing single-node approaches use full-fledged CPUs for expressive and performant execution of pointer-traversals [28, 170–172]. However, coupling large amounts of processing capacity with memory — which has utility in reducing data movement in PIM architectures [141–153] — 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 [182, 183] 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 (§**??**, [138]). Lastly, FPGA-based [138, 175, 184] and ASIC-based [168, 169] approaches achieve performance and energy efficiency by hard-wiring pointer traversal logic for specific data structures, limiting their expressiveness.

3.6 Summary

While PULSE is implemented atop Ethernet, its design is interconnect-agnostic and could be realized in ASIC-based or FPGA-attached memory devices over emerging interconnects like CXL [10, 184, 185]. We have verified these benefits in simulation atop detailed memory access and processing traces of our evaluated applications and workloads. The simulator maintains 2GB of cache in local

(CPU-attached) DRAM, while the entire working set is stored on remote CXL memory. Following prior work [37], we model 10–20ns L3 cache latency, 80ns local DRAM latency, 300ns CXL-attached memory latency, and 256B 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 hardware programmable Ethernet switch and FPGA accelerator latencies.

Fig. 3.12 shows the average slowdown for executing our evaluated workloads on CXL memory relative to running it completely locally (i.e., the entire application working set fits in local DRAM) — with and without PULSE. In the four-node setup, PULSE reduces CXL's slowdown by 19–33% across all applications.

In the single-node setup, PULSE still reduces the slowdown by 19–23% by minimizing high-latency traversals over the CXL interconnect. While a real hardware realization is necessary to precisely quantify PULSE's benefits, our simulation (which models the lowest possible CXL latency and highest possible PULSE overheads) highlights its potential for improving performance in emerging interconnects.

Chapter 4

Hardware Layer: Memory Management for Next-Gen Interconnects

While network-based resource disaggregation has gained attention due to advancements in network bandwidth (§**??**), the inherent latency, limited by the speed of light, still imposes significant overheads. This section explores the potential of next-generation interconnects and their impact on resource disaggregation.

4.1 Next-generation Interconnects

Recent advancements in hardware have led to the development of new-generation interconnects by major hardware vendors, such as NVLink [186] from Nvidia and Compute Express Link (CXL) [10] from Intel. CXL, in particular, has been introduced as a promising solution to expand memory capacity and bandwidth by attaching external memory devices to PCIe slots, offering a dynamic and heterogeneous computing environment.

Compute Express Link (CXL). As depicted in Figure **??**, CXL encompasses three key protocols: CXL.mem, CXL.cache, and CXL.io. CXL.io serves as the PCIe physical layer. CXL.mem enables processors to access memory over PCIe, while CXL.cache facilitates coherent memory access between processors and accelerators. These protocols allow for the construction of various CXL device types. The initial CXL 1.1 version serves as a memory expander for a single server. Subsequent versions, like CXL 2.0, extend this capability to multiple servers, incorporating CXL switches that coordinate access from different servers and enable various compute nodes to share a large memory pool. The forthcoming CXL 3.0 aims to scale up further, with cache coherency managed by hardware.

Despite extensive research on CXL [185, 187, 188], practical, commercial CXL hardware implementations remain in development, posing challenges in fully understanding performance and system support design for such hardware. Most studies have relied on simulations or FPGA-based CXL hardware [188, 189], lacking empirical evaluations on ASIC-based CXL hardware. Moreover, existing research often focuses on single aspects of CXL, like capacity or bandwidth, using synthetic benchmarks and neglecting a comprehensive evaluation that includes cost considerations. To gauge the performance of real CXL hardware and assess its suitability for resource disaggregation, we evaluated the latest hardware available: Intel's 4^{th} generation scalable processor (Sapphire Rapids) and Asteralabs's CXL 1.1 memory expander (Type-3 device). Using Intel Memory Latency Checker (MLC) [190], we measured the latency of reading data from the CXL device and local memory equipped with the same amount of DDR5 channels for local and cross-socket access. Figure**??** reveals that the latest CXL hardware exhibits a latency of more than $2.5\times$ higher than local memory. However, this gap narrows for cross-socket access, suggesting CXL as another memory tier. This raises questions about whether and how this information should be exposed to applications. Previous research [191] has investigated promoting hot pages from slower-tiered memory at the kernel level to enhance performance while maintaining application transparency.

This study represents the first available evaluation of real CXL 1.1 ASICs. The performance of CXL 2.0 and 3.0 remains to be explored in future work.

In an age marked by the surge of memory-intensive applications, such as machine learning tasks and High-Performance Computing (HPC) applications, there is an urgent need for expanding the memory capacity and bandwidth [192–194]. For instance, a machine learning application with 175 B model requires 700 GB of memory to hold its parameters only, not to mention memory requirements for intermediate results and others. That is, the memory requirements of modern applications could easily exceed the memory capability of a single machine due to physical constraints, such as availability of DDR DIMM slots and thermal issues, as well as cost considerations of employing high-density DIMMs [193, 194].

To meet such urgent demands, Compute Express Link (CXL) [10, 185, 187, 194] is introduced as

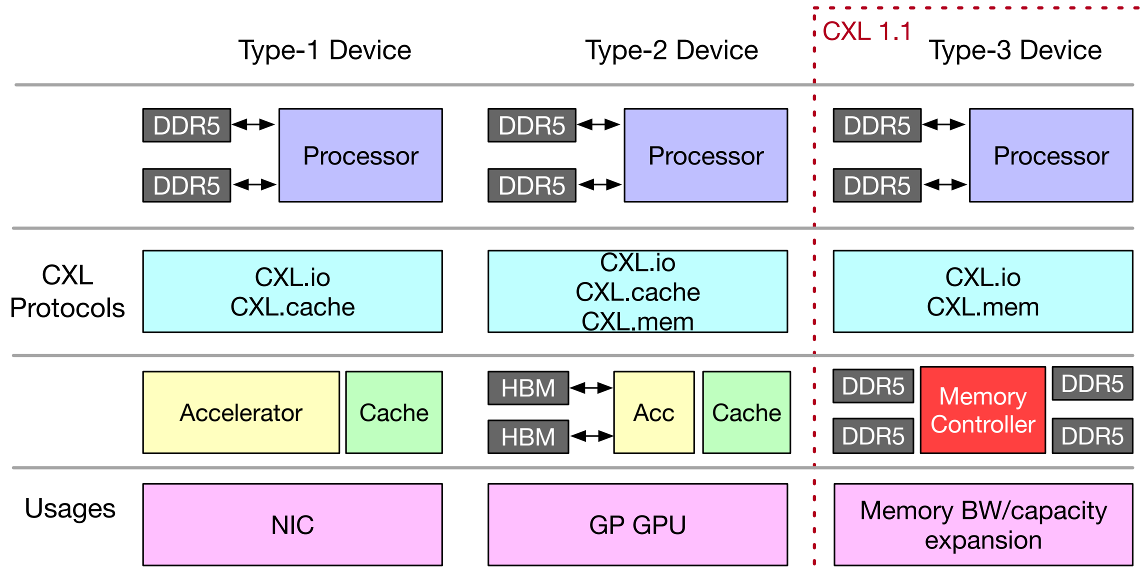


Fig. 4.1: **CXL Overview.** In this study, we focus on commercial CXL 1.1 Type-3 devices, leveraging CXL.io and CXL.mem protocols for memory expansion in single-server environments.

a groundbreaking interconnect technology. CXL promises significant expansion of memory capacity and bandwidth by attaching external memory devices (e.g., DRAM, Flash or persistent memory) to PCIe slots. Unlike its predecessors, CXL enables a more dynamic and heterogeneous computing environment, leading to various design trade-offs for performance and cost gains. Commercially debuting with version 1.1, CXL allows direct attachment of external memory devices to the host machine, enabling a unified and coherent memory address space. In such configuration, CXL is predominantly used as a way of memory expansion. For example, AsteraLabs' A1000 [195] CXL memory expansion card supports up to 4xDDR5 RDIMMs, enabling up to 2 TB of additional memory for a single server.

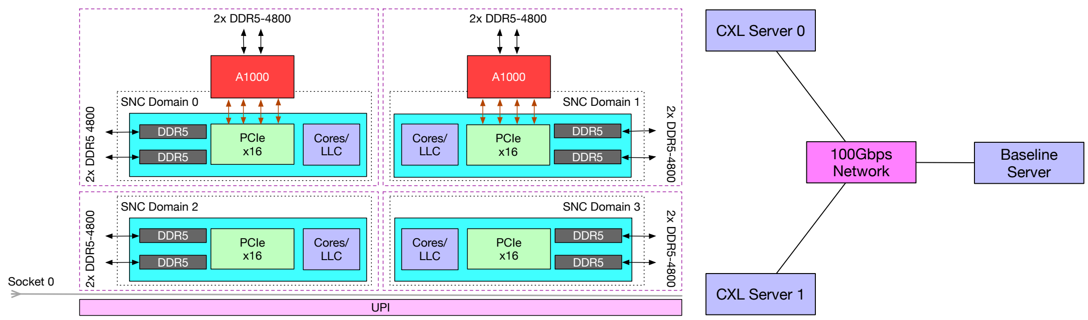
Although substantial studies on CXL memory have been performed in the past [37, 185, 187, 188, 191, 194, 196, 197], there remains a significant gap of employing these studies to guide the integration of CXL practically. In particular, we observe the following issues: (1) Much of the current literature has focused on evaluating CXL hardware through simulations [37, 187] or using FPGA-based setups [188, 197]. Although a limited number of studies have begun to assess the raw performance of ASIC-based CXL hardware [188, 198], there remains a gap in understanding how different system configurations influence the performance of data center applications using CXL memory. Furthermore, the specific applications that could substantially benefit from CXL memory expansion are not yet fully identified. (2) While existing studies have begun to explore the cost implications of employing CXL technology, such as the work on memory pooling cost models presented in [199], a critical gap remains in understanding the cost-effectiveness of migrating particular types of applications or services to memory expansions facilitated by CXL. (3) Given the restricted availability of CXL ASIC hardware, the research community faces a notable scarcity of open-source empirical data. This limitation hinders efforts to fully comprehend the performance capabilities of such hardware or to develop performance models based on empirical evidence.

Our study aims to fill existing knowledge gaps by conducting detailed evaluations of CXL 1.1 for memory-intensive applications, leading to several *intriguing observations*: Contrary to the common perception that CXL memory, due to its higher latency, should be considered a separate, slower tier of memory [37,191], **we find that shifting some workloads to CXL memory can significantly enhance performance**, even if local memory's capacity and bandwidth are underutilized. This is because using CXL memory can decrease the overall memory access latency by alleviating bandwidth contention on DDR channels, thereby improving application performance. From our analysis of application performance, we have formulated an abstract cost model (§4.6) that predicts substantial cost savings in practical deployments.

In summary, the major contributions of this paper are:

- **Empirical Evaluation of ASIC CXL Hardware**: Our study comprehensively examines the performance of ASIC-based CXL hardware and system configurations in data center applications, offering insights on optimizing CXL memory utilization.
- **Cost-Benefit Analysis**: We undertake a comprehensive cost-benefit analysis and develop an Abstract Cost Model to evaluate how CXL memory could substantially reduce real-world applications' TCO (Total Cost of Ownership).
- **Open-source data on CXL ASIC performance**: We open source all data and testing configurations under `https://github.com/bytedance/eurosys24-artifacts`.

The paper organizes as follows. §4.2 introduces basic information of CXL and environment setup for the evaluations. §4.3 presents basic performance characteristic of CXL memory expansion. §4.4 and §4.5 presents findings and suggestions of using CXL as the expansion of memory capacity and bandwidth on data center workloads. §4.6 provides a detailed analysis on the potential cost



(a) CXL Server(socket 0 illustrated here).

(b) Server setup.

Fig. 4.2: **CXL Experimental Platform.** (a) Each CXL server is equipped with two A1000 memory expansion cards. SNC-4(§4.3.1) is enabled only for the raw performance benchmarks(§4.3) and bandwidth-bound benchmarks(§4.5), and each SNC Domain is equipped with two DDR5 channels. (a) illustrates Socket 0; Socket 1 shares a similar setup except for the absence of CXL memory. (b) Our platform comprises two CXL servers and one baseline server. The baseline server replicates the same configuration but lacks any CXL memory cards.

benefits brought by CXL. §**??** discusses how our insights are applicable to future generations of CXL. §**??** describes related work, and §**??** concludes the paper.

4.2 Background and Methodology

This section presents an overview of CXL technology, followed by our experimental setup and methodologies.

4.2.1 Compute Express Link (CXL) Overview

Compute Express Link (CXL) [**?**] is a standardized interconnect technology that facilitates communication between processors and various devices, including accelerators, memory expansion units, and smart I/O devices. CXL is built upon the physical layer of PCI Express® (PCIe®) 5.0 [200], providing native support for x16, x8, and x4 link widths with data rates of 32.0 GT/s and 64.0 GT/s. The CXL transaction layer is implemented through three protocols: CXL.io, CXL.cache, and CXL.mem, as depicted in Fig. 4.1. *CXL.io* protocol is based on PCIe 5.0 and handles device discovery, configuration, initialization, I/O virtualization, and direct memory access (DMA). *CXL.cache* enables CXL devices to access the host processor's memory. *CXL.mem* allows the host to access memory attached to devices using load/store commands.

CXL devices are categorized into three types, each associated with specific use cases: (1) *Type-*

1 devices like SmartNICs utilize CXL.io and CXL.cache for DDR memory communication. (2) *Type-2 devices*, including GPUs, ASICs, and FPGAs, employ CXL.io, CXL.cache, and CXL.mem to share memory with the processor, enhancing various workloads in the same cache domain. (3) *Type-3 devices* leverage CXL.io and CXL.mem for memory expansion and pooling. This allows for increased DRAM capacity, enhanced memory bandwidth, and the addition of persistent memory without sacrificing DRAM slots. Type-3 devices complement DRAM with CXL-enabled solutions, benefiting high-speed, low-latency storage.

The commercially available version of CXL is 1.1, where a CXL 1.1 device can only serve as a single logical device accessible by one host at a time. Future generations of CXL, like CXL 2.0, are expected to support the partitioning of devices into multiple logical units, enabling up to 16 different hosts to access different portions of memory [201]. In this paper, our focus is on commercially available CXL 1.1 Type-3 devices, specifically addressing single-host memory expansion.

4.2.2 Hardware Support for CXL

Recent announcements have introduced CXL 1.1 support for Intel Sapphire Rapids processors (SPR) [202] and AMD Zen 4 EPYC "Genoa" and "Bergamo" processors [203]. While commercial CXL memory modules are provided by vendors such as Asteralabs [195], Montage [204], Micron [139], and Samsung [198], CXL memory expanders are predominantly in prototype stages, with only limited samples available, making access difficult for university labs. Consequently, due to the scarcity of CXL hardware, research into CXL memory has largely depended on NUMA-based emulation [37, 191] and FPGA implementations [188, 197], each with inherent limitations:

NUMA-based emulation. Given the cache coherent nature and comparable transfer speed of CXL and UPI/xGMI interconnects, NUMA-based emulation [37, 191] is widely adopted to enable fast application performance analysis and software prototyping as the CXL memory is exposed as a remote NUMA node. However, NUMA-based emulation fails to accurately capture the performance characteristics of CXL memory due to differences from CXL and UPI/xGMI interconnects [205], as shown in previous research [188].

FPGA-based implementation. Intel and other hardware vendors use FPGA hardware to implement CXL protocols [189], bypassing the performance inconsistencies of NUMA-based emulation.

However, FPGA-based CXL memory falls short in fully utilizing memory chip performance due to its lower operating frequency compared to ASICs [206]. FPGAs prioritize flexibility over performance and are suitable for early-stage CXL memory validation but not production deployment. Intel's recent evaluation [188] uncovered performance issues in FPGA implementations, including reduced memory bandwidth during concurrent thread execution. This hampers rigorous evaluations for memory capacity- and bandwidth-bound applications, which are key use cases for CXL memory expanders. Further discussion on the performance disparity between CXL ASIC and FPGA controllers is in §4.3.

To the best of our knowledge, we are one of the pioneers in uncovering the performance characteristics of actual ASIC prototypes designed for CXL memory expansion. The ASIC CXL memory controller we have employed is the A1000 [195] developed by AsteraLabs, which implements the CXL interface at speeds of up to 32 GT/s per lane, supporting up to 16 lanes in total. This controller has the capability to accommodate up to 4 DDR5-5600 RDIMM slots, providing a total memory capacity of 2TB.

4.2.3 Software Support for CXL

While hardware vendors are actively advancing CXL production, a notable deficiency exists in software and OS kernel support for CXL memory. This deficiency has prompted the utilization of specific software enhancements. We summarize the most recent patches in the Linux Kernel that add CXL-aware support, namely: (1) the interleaving policy support (unofficial) and (2) the hot page selection support (official since Linux Kernel v6.1).

N:M Interleave Policy for Tiered Memory Nodes.

Traditional memory interleave policies distribute data evenly across memory banks, often using a 1:1 ratio. However, the advent of tiered memory systems, which feature CPU-less memory nodes with diverse performance traits, demands more nuanced strategies for optimizing memory bandwidth, especially for bandwidth-heavy applications. The interleave patch [207] introduces an innovative N:M interleave policy to address this, allowing for an allocation scheme where N pages are directed to high-performance (top-tier) nodes and M pages to lower-tier nodes. For example, using a 4:1 ratio directs 80% of traffic to top-tier nodes and 20% to low-tier nodes, adjustable through the `vm.numa_tier_interleave` parameter. While the patch showcases compelling evaluation results [207], it's crucial to note that optimal memory distribution depends on specific hardware and application characteristics. Given the higher latency of CXL memory, as demonstrated in §4.3, performance-sensitive applications should undergo thorough profiling and benchmarking to maximize the advantages of interleaving and mitigate potential performance trade-offs.

NUMA Balancing & Hot Page Selection.

The memory subsystem, now termed a memory tiering system, accommodates various memory types like PMEM and CXL Memory, each with differing performance characteristics. To optimize system performance, "hot pages" (frequently accessed) should reside in faster memory tiers like DRAM, while "cold pages" (less frequently accessed) should be in slower tiers like CXL memory. Recent Linux Kernel patches address this:

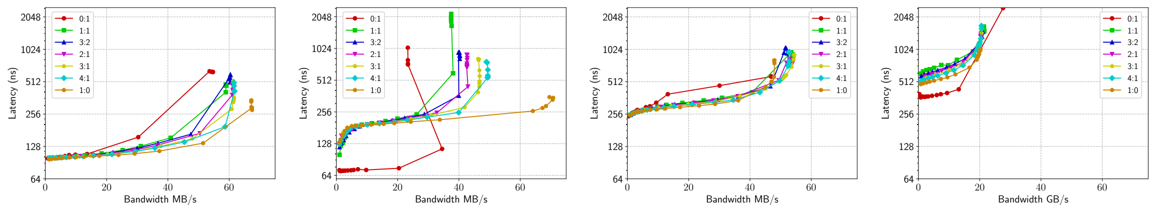
1. The *NUMA-balancing* patch [208] uses a latency-aware page migration strategy, focusing on promoting recently accessed pages (MRU). It scans NUMA balancing page tables and hints page faults. However, it may not accurately identify high-demand pages due to extended scanning intervals, potentially causing latency issues for some workloads.

2. The *Hot Page Selection* patch" [**?**] introduces a Page Promotion Rate Limit (RPRL) mechanism to control the rate of page promotions and demotions. While this extends promotion/demotion times, it improves workload latency. The hot page threshold is dynamically adjusted to align with the promotion rate limit.

Additionally, research prototypes like TPP [191] share a similar concept with optimizations and are being considered for integration into the Linux Kernel [209]. However, we faced challenges with TPP when running memory-bandwidth-intensive applications, resulting in unexplained performance degradation. Hence, we rely on the well-tested kernel patches integrated into Linux Kernel since version 6.1.

4.2.4 Experimental Platform Description

The evaluation testbed, as illustrated in Fig. 4.2(b), consists of three servers. Two of these servers are designated as CXL experiment servers. Each of these servers is equipped with dual Intel Xeon 4th Generation CPUs (Sapphire Rapids, or SPR), 1 TB of 4800 MHz DDR5 memory, two 1.92 TB



(a) Local-socket MMEM (b) Remote-socket MMEM (c) Local-socket CXL (d) Remote-socket CXL

Fig. 4.3: **Overall effect of read-write ratio on MMEM and CXL across different distances.** The workloads are represented by read:write ratios (e.g., 0:1 for write-only, 1:0 for read-only). Accessing CXL memory locally incurs higher latency compared to MMEM but is more comparable to accessing MMEM on a remote socket. MMEM bandwidth peaks at 67 GB/s, versus 54.6 GB/s for CXL memory. Performance significantly declines when accessing CXL memory on a remote socket (§4.3.2). In specific scenarios, such as the write-only workload (0:1) in (b), the plot may show instances where bandwidth decreases and latency increases with heavier loads. The Y-axis is on a logarithmic scale.

SSDs, and a pair of A1000 CXL Gen5 x16 ASIC memory expanders modules from AsteraLabs, each with 256 GB of 4800MHz memory (resulting in a total of 512 GB memory per server). Both A1000 memory modules are attached to socket 0. The third server serves as the baseline and is configured identically to the CXL experiment servers, except for the absence of the CXL memory expanders. It is designated for initiating client requests and running workloads that strictly utilize the main memory during the application assessments. All servers are interconnected via 100 Gbps Ethernet links.

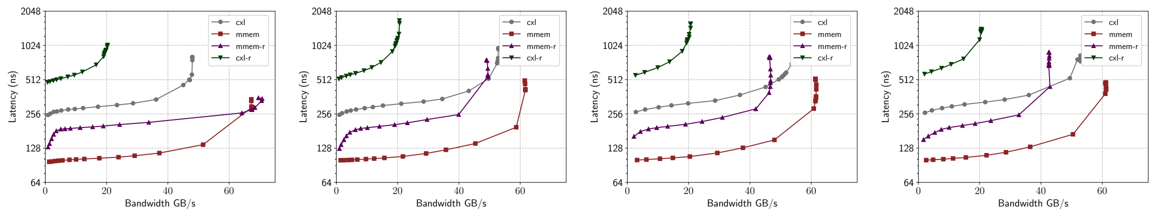
4.3 CXL 1.1 Performance Characteristics

In this section, we assess the performance of the CXL memory expander and compare it directly with main memory, which we designate as **MMEM** for clarity against CXL memory. We analyze workload patterns and evaluate performance differences between local and remote socket scenarios.

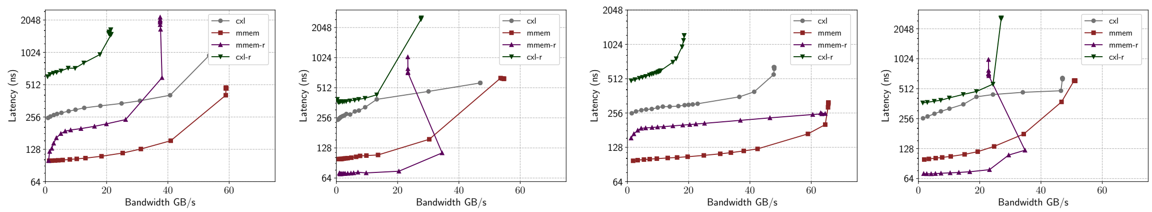
4.3.1 Experimental Configuration

For each dual-channel A1000 ASIC CXL memory expander [195], we connect two DDR5-4800 memory channels, achieving a total capacity of 256 GB. To provide a fair comparison between MMEM and CXL-attached DDR5 memory, we utilize the Sub-NUMA Clustering (SNC) [210] feature to ensure the number of memory channels is the same in both settings.

Sub-NUMA Clustering(SNC). Sub-NUMA Clustering (SNC) serves as an enhancement over the



(a) Read-only workload (b) Read:Write = 4:1 workload (c) Read:Write = 3:1 workload (d) Read:Write = 2:1 workload



(e) Read:Write = 1:1 workload (f) Write-only workload (g) Read-only workload (random) (h) Write-only workload (random)

Fig. 4.4: **A detailed comparison of MMEM versus CXL over diverse NUMA/socket distances and workloads.** (a)-(f) shows the latency-bandwidth trend difference of accessing data from different distances in sequential access pattern, sorted by the proportion of write. We refer to main memory as **MMEM**, with MMEM-r and CXL-r representing remote socket MMEM and cxl memory access, respectively. The Y-axis is on a logarithmic scale.

traditional NUMA architecture. It decomposes a single NUMA node into multiple smaller semi-independent sub-nodes (domains). Each sub-NUMA node possesses its own dedicated local memory, L3 caches, and CPU cores. In our experimental setup (Fig. 4.2(a)), we partition each CPU into four sub-NUMA nodes. Each sub-NUMA node is equipped with two DDR5 memory channels connected to two 64 GB DDR5-4800 DIMMs. Enabling SNC requires setting the IMC (Integrated Memory Controllers) to 1-way interleaving. According to the specifications, a single DDR5-4800 channel has a theoretical peak bandwidth of 38.4 GB/s [187]. Therefore, each sub-NUMA node has a combined memory bandwidth of up to 76.8 GB/s.

Intel Memory Latency Checker (MLC). We leverage Intel's Memory Latency Checker (MLC) to examine loaded-latency for various read-write workloads, adopting a 64-byte access size same as prior work [188]. We deploy 16 MLC threads, and it's important to note that while the thread count is a configurable parameter in MLC, it doesn't directly dictate memory request concurrency. MLC assigns separate memory segments for each thread to access simultaneously. Specifically, when evaluating loaded latency, MLC incrementally increases the operation rate of each thread. Our findings indicate that employing 16 threads with MLC precisely measures both the idle and loaded latency and the point at which bandwidth becomes saturated. MLC accommodates a broad spectrum of workloads including those with varied read-write mixes and non-temporal writes.

Our study is focused on addressing the following research questions:

- How is the performance of the CXL-attached memory compared to that of local-socket/remote-socket main memory?
- What is the performance impact of the CXL memory under different read-write ratios and access patterns (random vs. sequential)?
- How do main memory and CXL memory behave under high memory load conditions?

4.3.2 Basic Latency and Bandwidth Characteristics

This section outlines our findings on memory access latency and bandwidth for different memory configurations: local-socket main memory (MMEM), remote-socket main memory (MMEM-r), CXL memory (CXL), and remote-socket CXL memory (CXL-r). Figure 4.3(a) shows the loaded latency curve for MMEM under varied read-write mixes. The read-only workload hits a peak bandwidth of roughly 67 GB/s, reaching 87% of its theoretical maximum. Yet, as write operations increase, bandwidth dips, with write-only tasks dropping to 54.6 GB/s. We note an initial memory latency of about 97 ns, which spikes exponentially as bandwidth nears full capacity, a sign of bandwidth contention [211, 212]. Interestingly, latency starts to significantly increase at 75%-83% of bandwidth utilization, surpassing prior estimates of 60% from earlier studies [211].

Figure 4.3(b) illustrates the latency differences when accessing MMEM via a remote socket. For read-only tasks, latency begins at approximately 130 ns, contrasting sharply with just 71.77 ns for write-only operations. This reduced latency for write-only workloads results from non-temporal writes, which proceed asynchronously without awaiting confirmation. Despite read-only tasks achieving maximum bandwidth comparable to that of local MMEM, incorporating more write operations significantly diminishes bandwidth, attributed to the additional UPI traffic necessitated by cache coherence protocols. Interestingly, the write-only workload generate minimal UPI traffic but suffer the lowest bandwidth as it utilize only one direction of the UPI's bidirectional capabilities. Moreover, latency escalation occurs earlier in remote socket memory accesses than in local ones, primarily due to queue contention at the memory controller.

Fig. 4.3(c) illustrates the latency curve for CXL memory expansion, demonstrating a minimum latency of 250.42 ns. Interestingly, despite additional PCIe and CXL memory controller overhead on the datapath, accessing CXL follows the same "Bandwidth contention" trend as MMEM. The latency of accessing CXL on the same socket remains relatively stable as bandwidth increases, with a maximum bandwidth of around 56.7 GB/s, achieved when the workload is 2:1 read-write ratio. The reduction in maximum bandwidth compared to DRAM is attributed to PCIe overhead, such as extra headers. The maximum bandwidth for read-only workloads is smaller due to PCIe bi-directionality, preventing full bandwidth utilization. Fig. 4.3(d) reveals the latency-bandwidth plot for accessing CXL from a remote socket, incurring an exceptionally high idle latency of 485 ns. In addition, the maximum memory bandwidth is unexpectedly halved, reaching just 20.4 GB/s for 2:1 read-write ratio, which is a much more severe performance drop compared to accessing MMEM from the remote NUMA node in Fig. 4.3(d). Since running a read-only towards a CXL Type-3 device on the remote socket does not generate substantial coherence traffic, initial speculation regarding cache coherence is ruled out. Further investigation utilizing the Intel Performance Counter Monitor (PCM) [213] also confirms that the UPI utilization is consistently below 30%. Discussions with Intel suggest this performance bottleneck is likely due to limitations in the Remote Snoop Filter (RSF) on the current CPU platform, anticipated to be addressed in the next-generation processors [214].

4.3.3 Different Read-Write Ratios & Access Pattern

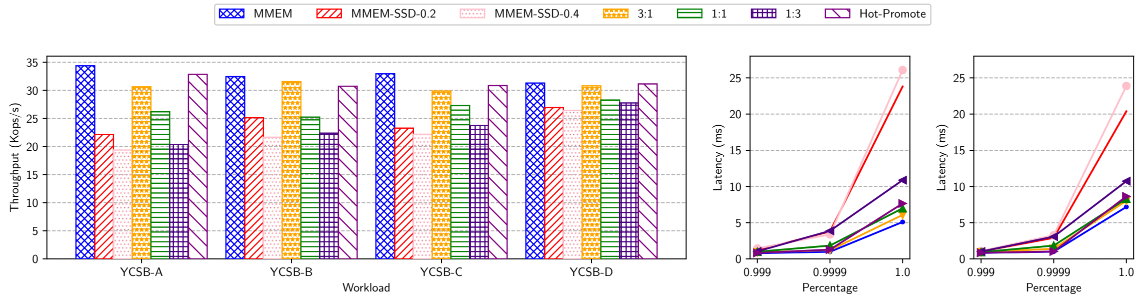
Fig. 4.4(a)-4.4(f) present a performance comparison for a specific workload with varying read-write ratios. The results align with our observation that accessing CXL from a remote socket introduces exceptionally high latency and low bandwidth. When accessing CXL from the same socket, latency is 2.4-2.6 $\times$ that of local DDR and 1.5-1.92 $\times$ that of remote socket DDR. This suggests that running applications directly on CXL may significantly drop performance. However, when workloads span multiple NUMA nodes within the same socket, accessing CXL locally is comparable to accessing remote NUMA node memory. Additionally, the latency-bandwidth knee-point shifts to the left as the proportion of write operations in the workload increases. Fig. 4.4(g) and 4.4(h) display the results of running both read-only and write-only workloads, utilizing random access patterns instead of sequential access. Notably, we do not observe any significant performance disparities under these conditions.

4.3.4 Key insights

Avoiding Remote Socket CXL Access. CXL memory expansion is commonly utilized for applications that are demanding in terms of memory, particularly those limited by memory capacity or bandwidth. In such contexts, accessing memory across sockets is not uncommon. It is important for software developers to recognize the potential decline in performance when CXL memory is accessed from a remote socket and to strategize against cross-socket CXL memory accesses in their applications. Additionally, hardware vendors should perform cooperative testing and validation of their products to ensure compatibility between CXL memory modules and the processors' CXL support. With adequate support for the CXL 1.1 protocol, we expect that the maximum bandwidth attainable when accessing CXL memory across sockets could approximate the bandwidth seen when accessing MMEM across sockets.

Bandwidth Contention Previous research [187, 212] has brought attention to issues related to bandwidth contention. We further examine how memory latency varies with varying read-write ratios under bandwidth contention. While latency remains relatively stable at low to moderate bandwidth utilization levels, it increases exponentially as bandwidth approaches higher levels, primarily due to queuing delays in the memory controller [211]. Furthermore, the knee-point in latency shifts to lower memory bandwidth when there is a higher proportion of write operations in the workload. Interestingly, CXL-attached memory has often been characterized by industry and research community as 'tiered memory' [188, 207, 209], suggesting that it serves as a slower and less performant memory layer to be considered only when MMEM is fully utilized. However, we argue against this simplistic view of CXL-memory. Allocators and kernel-level page placement policies should consider the available bandwidth in MMEM. Even if a substantial portion of memory bandwidth in MMEM remains unused, e.g., 30%, offloading a portion of the workload, e.g., 20%, to CXL memory can lead to overall performance improvements. Our recommendation is to regard CXL memory as a valuable resource for load balancing, even when local DRAM bandwidth is not fully utilized. Subsequent real-world evaluations support these insights (§4.5).

Comparison with FPGA-based CXL implementations. Intel recently disclosed latency and bandwidth performance metrics for their FPGA-based CXL prototype [188]. While they provided



(a) Throughput of YCSB workload (b) Tail latency of YCSB-A (c) Tail latency of YCSB-C

Fig. 4.5: **KeyDB YCSB latency and throughput under different configurations.** (a) Average throughput of four YCSB workload under different system configuration. (b) Tail latency of YCSB-A (c) Tail latency CDF of YCSB-C, both reported by the YCSB client [215].

insights into relative latency and bandwidth efficiency for soft and hard IP implementations, performance under load was not shared. Our measurements indicate that the ASIC CXL solution only introduces a less than $2.5x$ overhead in access latency compared to MMEM, surpassing most of Intel's measurements. However, the FPGA-based solution achieved only 60% of the PCIe bandwidth due to the inefficiency of the memory controller, while the Asteralabs A1000 prototype reached an impressive 73.6% bandwidth efficiency, clearly outperforming Intel's FPGA-based solution.

4.4 Memory Capacity-bound Applications

One of the most significant advantages of integrating CXL memory into modern computing systems is the opportunity for significantly larger memory capacities. To elucidate the potential benefits, we focus on three particular use cases (1) key-value stores, a commonly used application in data centers. (2) Big data analytical application. (3) Elastic computing from cloud providers.

4.4.1 In-memory key-value stores

Redis [24] is an open-source in-memory key-value store and one of the most popular NoSQL databases. Redis employs a user-defined parameter, `maxmemory`, to limit its memory allocation for storing user data. Like traditional memory allocators (e.g., malloc()), Redis may not return memory to the system after key deletion, particularly if deleted keys were on a memory page with active ones. This necessitates memory provisioning based on peak demand, making memory capacity the major bottleneck for Redis deployments [216] in data centers. Google Cloud suggests keeping memory usage below 80% [217], whereas other sources recommend a limit of 75% [216].

Due to the substantial infrastructure costs for memory-only deployment, Redis Enterprise [218] is the commercial variant extensively supported by leading cloud platforms (e.g., AWS, Google Cloud, or Azure). It introduces "Auto Tiering" [219] to allow data overflow to SSDs, offering an economically viable option for database expansion beyond the limits of RAM capacity. Given that Redis Enterprise is not accessible on our experiment platform, we employ KeyDB as an alternative. KeyDB extends Redis's capabilities by adding KeyDB Flash, which uses RocksDB for persistent storage. The FLASH feature enables all data is written to the disk for persistence, with hot data remaining in memory as well as disk.

Methodology and Software Configurations.

In our study, we investigate the performance effects of maximizing memory utilization on a KeyDB server. We deploy a single KeyDB instance on a CXL-enabled server configured with seven *server-threads*. Unlike Redis's single-threaded approach, KeyDB enhances performance by operating multiple threads to run the standard Redis event loop, akin to running several Redis instances simultaneously. We disable SNC and Transparent Hugepages and enable memory overcommitting within the kernel to minimize potential overhead from OS configurations. For KeyDB FLASH, we deactive all forms of compression in RocksDB to minimize software overhead. Our empirical analysis uses the YCSB benchmark with four distinct workloads: (1) YCSB-A (50% read, 50% update) for update-intensive scenarios; (2) YCSB-B (95% read, 5% update) for read-heavy operations; (3) YCSB-C (100% read) for read-only tasks; and (4) YCSB-D (95% read, 5% insert) to simulate reading the most recent data. These workloads are tested under various system configurations as detailed in Table 4.1. Note that we use the term "MMEM" for main memory in order to separate it from CXL memory. For configurations utilizing SSD data spillover, we set the *maxmemory* parameter according to the portion of the workload expected to remain in memory. For Hot-Promote, we applied *numactl* to distribute half of the dataset across CXL memory while limiting the total main memory usage to half the dataset size. The experiments are conducted using a 1 KB key-value size, the YCSB default, with a Zipfian distribution for workloads A-C and the latest distribution for workload D. The total amount of working set data is 512 GB.

Configuration	Description
`MMEM`	Entire working set in main memory.
`MMEM-SSD-0.2`	20% of the working set is spilled to SSD.
`MMEM-SSD-0.4`	40% of the working set is spilled to SSD.
`3:1`	Entire working set in memory (75% MMEM + 25% CXL, 3:1 interleaved).
`1:1`	Entire working set in memory (50% MMEM + 50% CXL, 1:1 interleaved).
`1:3`	Entire working set in memory (25% MMEM + 75% CXL, 1:3 interleaved).
`Hot-Promote`	Entire working set in memory (50% MMEM + 50% CXL), with hot page promotion kernel patches discussed in §4.2.

Table 4.1: **Configurations used in capacity experiments.**

Analysis.

Fig. 4.5 provides insights into the variations in throughput across different configurations. Notably, regardless of the specific workload, running the entire workload on MMEM consistently yields the highest throughput. This outcome can be attributed to the nature of our workload, primarily constrained by memory capacity rather than memory bandwidth. The Hot-Promote configuration, which leverages the Zipfian distribution to identify frequently accessed keys as hot pages and migrates them from CXL to MMEM, performs nearly as well as running the workload entirely on MMEM. This demonstrates the effectiveness of the Hot-Promote approach in optimizing performance. In contrast, interleaving data access between CXL and MMEM leads to a noticeable performance decrease, resulting in a 1.2x to 1.5x slowdown compared to running the workload directly in MMEM. This performance drop is primarily due to the higher access latency, as evident in the tail latency plots for workload A and workload C (Fig. 5(b)(c)). MMEM-SSD-0.2 and MMEM-SSD-0.4 configurations perform the poorest, exhibiting nearly a 1.8x slowdown compared to the pure MMEM solution and a 1.55x slowdown compared to the CXL interleaving solution. This poor performance is mainly attributed to the high access latency required to retrieve data from the SSD. It's worth noting that our choice of a Zipfian distribution ensures that the working set is largely cached in MMEM. If the keys were distributed uniformly, we anticipate worse performance due to increased SSD access times.

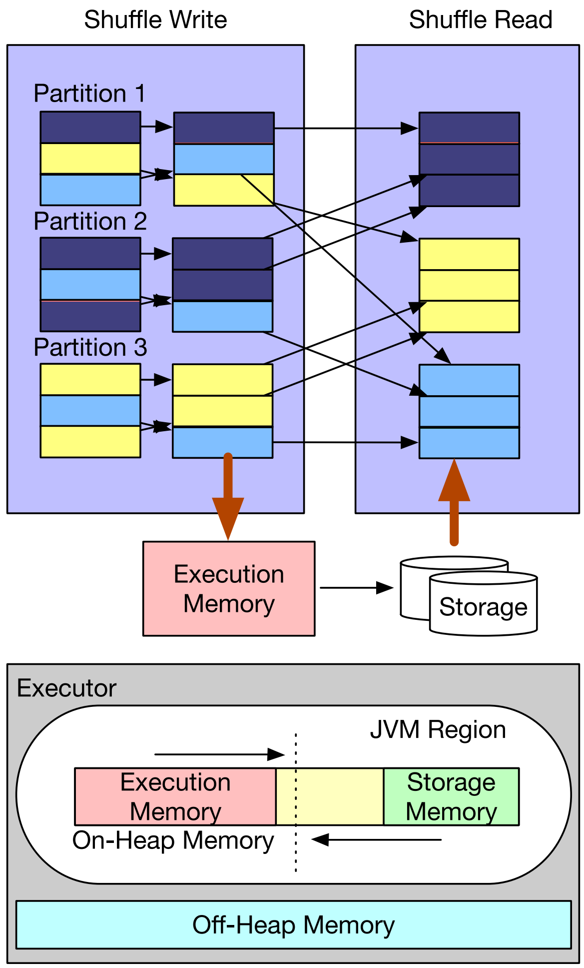


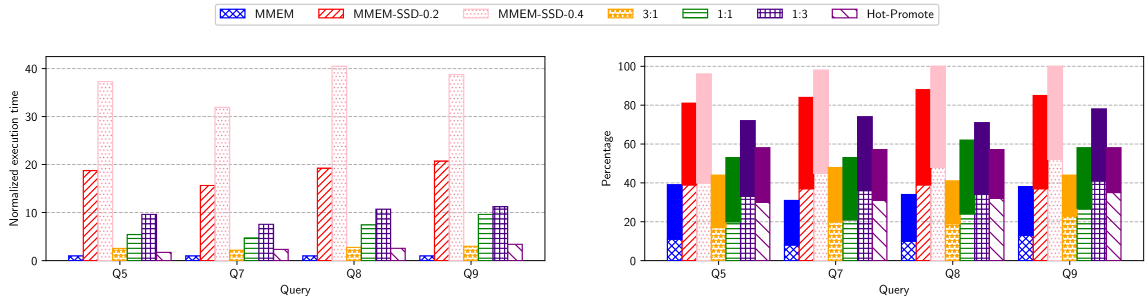
Fig. 4.6: **Spark memory layout and shuffle spill.** Each Spark executor possesses a fixed-size On-Heap memory, which is dynamically divided between execution and storage memory. If there is insufficient memory during shuffle operations, the Spark executor will spill the data to the disk.

Insights.

Our study shows that the additional memory capacity provided by CXL can be a game-changer for applications like key-value stores constrained by traditional MMEM's capacity. Intelligent scheduling policies further accentuate the benefits, offering avenues for optimizing systems that leverage multiple memory types and simultaneously saving operation costs.

4.4.2 Spark SQL

Big Data plays a crucial role in the workloads managed by data centers. Due to the scale of data involved in Big Data analytical applications, memory capacity often becomes a bottleneck to the performance [26]. Take Spark [67], one of the common Big Data platforms, as an exam-



(a) Normalized execution time of TPC-H queries

(b) Percentage of time spent on shuffle r/w

Fig. 4.7: **Spark execution time and shuffle percentage.** (a) Execution time of each TPC-H query normalized to the execution time running on MMEM. (b) The percentage of time spent of shuffle operation for each query. The solid bars represent shuffle writes, while hollow bars represent shuffle reads.

ple: A typical query requires shuffling data from multiple tables for processing in the next stage. Operations like *reduceByKey()* first partition the data according to the key and then execute reduce operators on each key. Such shuffling operation involves disk I/O and network communication between multiple nodes, posing significant overhead on the query. In some cases, the performance of shuffling could dominate the performance of the workload [220]. During the shuffling process(Fig. 4.6), memory usage could grow beyond the capacity or certain threshold (e.g. `spark.shuffle.memoryFraction`). When this happens, Spark can be configured to spill data to disk to avoid the risk of out-of-memory failure. Since disk I/O is of magnitudes slower than memory, this could significantly impact the workload's performance.

Methodology and Software Configurations.

In our experiment, we aim to test if we could reduce the number of servers needed for a specific workload with minimal effect on overall performance. Therefore, we compared the performance of Spark running TPC-H [221] on three servers without CXL memory expansion vs. on two servers but with CXL memory expansion. We assume the maximum amount of MMEM that could be used on each server is 512 GB, therefore with three servers, we have 1.5 TB MMEM and 1 TB CXL memory in total. In order to trigger data spill within the workload, we configured 150 Spark executors. Each Spark executor contains 1 core and 8 GB of memory. Therefore the total Spark application occupies 150 cores and 1.2 TB of memory. We generate a total of 7 TB TPC-H initial dataset. We continue to adhere to the configuration settings detailed in Table 4.1 as follows:

- MMEM only: We allocate 50 Spark executor and 400 GB on each of the **three** servers. In this case there is no data spilled to disk as each executor have sufficient amount of memory.

- MMEM/CXL interleaving: We distributed the same number of executors (150) across the **two** cxl servers, which has 1 TB (512 GB from each of the two CXL cards) plus 1 TB of MMEM (512 GB each). For example, in a configuration where MMEM and CXL memory usage is balanced (1:1 ratio), we allocated 75 Spark executors to use 600 GB MMEM while another 75 Spark executors to 600 GB CXL memory. In this case, there is also negligible amount of data spilled to the disk.

- Spill to SSD: To simulate conditions where executors would run out of memory and need to spill data to SSD storage, we restrict the memory allocation of the Spark executors to either 80% or 60% of entire 1.2 TB MMEM. In this case, there will be around 320 GB and 500 GB data spilled to the disk respectively.

- Hot-Promote: same as prior experiment (§4.4.1).

We chose four specific queries ($Q5$, $Q7$, $Q8$, and $Q9$) from the TPC-H benchmark [221], recognized for their intensive data shuffling demands from prior studies [220], to evaluate our setup. Importantly, our measurements focused solely on the time to execute these queries, excluding any data preparation or server setup durations. We disabled SNC on all servers.

Analysis.

Figure 4.7 illustrates variations in total execution time across different configurations. To provide a clear comparison, we normalized the total execution time against the best-case scenario, which involves running the entire workload in MMEM. Similar to the KeyDB experiments, the interleaving approach still exhibits a performance slowdown, ranging from 1.4x to 9.8x compared to the optimal MMEM-only scenario while using less number of servers. This performance degradation becomes worse as a larger proportion of memory is allocated to CXL. Nevertheless, it's crucial to note that even with this slowdown, the interleaving approach remains significantly faster than spilling data to SSDs. Figure 4.7(b) illustrates that shuffling overshadows the total execution time due to the intensification of data spill issues.

Year	CPU	Max vCPU per server	Memory channels per socket	Max memory \TB	Required Memory (1 : 4) \TB
2021	IceLake-SP [222]	160	8xDDR4-3200	4	0.64
2022 (delayed)	Sapphire Rapids [223]	192	8xDDR5-4800	4	0.768
2023 (delayed)	Emerald Rapids [224]	256	8xDDR5-6400	4	1
2024+	Sierra Forest [225]	1152	12	4	4.5
2025+	Clearwater Forest [226]	1152	TBD	4	4.5

Table 4.2: **Intel Processor Series.**

A notable difference between the KeyDB and Spark experiments is the performance of Hot-Promote. While it performs better in KeyDB, the Spark SQL experiment shows a more than 34% slowdown compared to MMEM. Unlike the Zipfian distribution in which the hottest keys are moved from CXL to DDR, there is a considerable amount of thrashing behavior within the kernel in the Spark SQL tests. We identify the root cause after thoroughly investigating the kernel patch implementation. In the initial version of the hot page selection patch [**?**], a sysctl knob "`kernel.numa_balancing_promote_rate_limit_MBps`" is used to control the maximum promoting/demoting throughput. Subsequent versions introduced an automatic threshold adjustment feature to this patch, aiming to strike a balance between the speed of promotion and migration costs. Nevertheless, this automatic adjustment mechanism appears to fall short in our Spark SQL evaluations. The TPC-H workload on Spark, which demonstrates reduced data locality, challenges the kernel's efficiency in promoting frequently accessed pages. This finding aligns with similar issues highlighted in prior research [188].

Insights.

Our research indicates that utilizing CXL memory expansion offers a cost-efficient approach for data-center applications. We postpone our detailed theoretical examination of the Abstract Cost Model to §4.6. Concurrently, although the hot-promote patch demonstrates significant advantages in key-value store workloads, its performance is notably lacking in Spark experiments. As system developers begin to enhance software support for CXL within the kernel, it is crucial to proceed with caution. System-wide policies can have varied impacts on applications, depending on their unique characteristics.

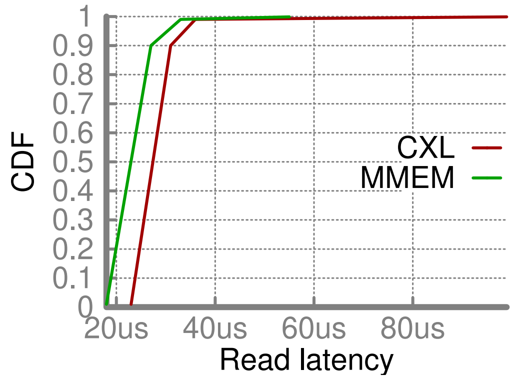
4.4.3 Spare Cores for Virtual Machine

One widely-used application within Infrastructure-as-a-Service (IAAS) is Elastic Computing [227]. Here, cloud service providers (CSPs) offer computational resources to users through virtual machines or container instances. Given the diverse needs of users, CSPs traditionally offer a variety of instance types, each characterized by different configurations of CPU cores, memory, disk, and network capacities. Generally, an "optimal" CPU-to-memory ratio, often cited as 1:4, is employed to balance computational and memory requirements (as per AWS guidelines [228, 229]). For example, an instance with 128 vCPUs would typically feature 512 GB of DDR memory. Advancements in server processor architecture and chiplet technology have spurred rapid increases in the number of cores available in a single processor package, driven in large part by the CSPs' aim to lower per-core costs. Consequently, 2-socket servers have seen their vCPU counts grow from 160 to 256 within the past two years (Table 4.2). This trend is projected to continue, reaching as many as 1152 vCPUs per server by 2025.

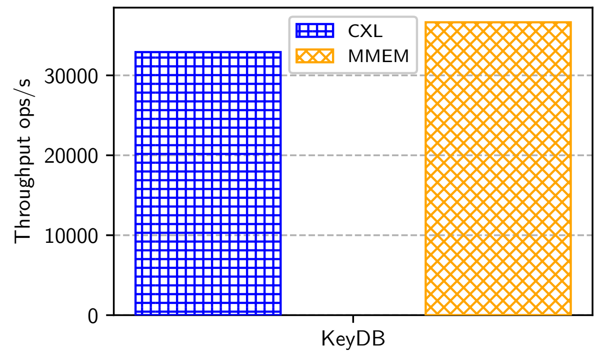
The surge in vCPUs exacerbates memory capacity bottlenecks, constrained by DDR slot limits, DRAM density, and the cost of high-density DIMMs. Intel's Sierra Forest Xeon, for example, supports 1152 vCPUs but is limited by motherboard design to less than 4 TB of memory, falling short of the typical 4.5 TB needed for VM provisioning [230]. This discrepancy makes maintaining a cost-effective vCPU-to-memory ratio challenging, resulting in underutilized vCPUs and lost revenue for CSPs. CXL memory expansion provides a solution by enabling memory capacity to scale beyond DDR limitations, ensuring optimal vCPU utilization and mitigating revenue losses for CSPs.

Methodology and Software Configurations.

To assess the performance impact when an application operates exclusively on CXL memory, we replicate the KeyDB configuration from previous experiments (§4.4.1). We utilize *numactl* to allocate the KeyDB instance exclusively to MMEM or CXL memory. For our evaluation, the workload employed is YCSB-C, characterized by 1 KB key-value pairs and a total dataset size of 100 GB. SNC is disabled.



(a) CDF of KeyDB YCSB-C



(b) Throughput of KeyDB YCSB-C

Fig. 4.8: **KeyDB Performance with YCSB-C on CXL/MMEM.**

Analysis.

The CDF of read latency (Fig. 4.8(a)) indicates that applications running on CXL experience a latency penalty of $9\% - 27\%$ which is less than the raw data fetching numbers in our previous measurements in §4.3. This is due to the processing latency within Redis. The throughput of running the entire workload on CXL memory is around 12.5% less compared to MMEM as show in Fig. 4.8(b).

Now consider a server operating at a sub-optimal vCPU-to-memory ratio of 1:3: (1) Due to inadequate memory, only 75% of the vCPUs can be sold at the optimal 1:4 ratio, resulting in a 25% revenue loss. Implementing CXL memory expansion enables the CSP to sell the remaining 25% of vCPUs at the optimal ratio. (2) Our benchmarks indicate that instances running on CXL memory perform 12.5% slower than those on DDR for common workloads such as Redis. Assuming a 20% price discount on such instances, CSPs could still recover approximately 80% of the lost revenue, equating to a 27% improvement in total revenue ($20/75 = 26.77\%$).

Insights.

Given the sheer scale of Elastic Computing Service (ECS) applications in public clouds, the potential benefits of CXL memory expansion could be substantial. However, the challenge of maintaining an optimal virtual CPU (vCPU) to memory ratio, traditionally at 1:4, becomes more complex with the rapid increase in processor cores. This ratio, although standard, is under scrutiny for its applicability in future cloud computing paradigms. Notably, Bytedance's Volcano Engine Cloud [231] illustrates the variability in resource allocation by offering different ratios: 1:4 for general purposes, 1:2 for compute-intensive tasks, and 1:8 for memory and storage-intensive workloads. The impact of CXL memory expansion and pooling on these established ratios presents an intriguing avenue for exploration, raising questions about the adaptability of cloud providers to evolving hardware capabilities and the subsequent effect on resource allocation standards.

4.5 Memory Bandwidth-Bound applications

The other advantage of CXL memory expansion is its extra memory bandwidth. We use Large Language Model inference as an example to showcase how this can benefit real-world applications.

Recent work on LLM [232] shows that LLM inference is hungry for memory capacity and bandwidth. The limited capacity of GPU memory restricts the batch size of the LLM inference job and reduces computing efficiency since LLM models are memory-demanding. On the other hand, while CPU memory is high in capacity, it has lower bandwidth than GPU memory. The extra bandwidth and capacity offered by CXL memory make it a promising option for alleviating this bottleneck. For example, a CPU-based LLM inference job can benefit from the extra bandwidth brought by CXL memory, and a CXL-enabled GPU device can also use the extra memory capacity from a disaggregated memory pool. Due to the lack of CXL support in current GPU devices, we experiment with LLM inference on CPU to study the implications of CXL memory's extra bandwidth. We also note that as LLM inference applications are agnostic to the underlying memory technologies, the findings and implications from our experiments are also applicable to the upcoming CXL 2.0/3.0 devices.

LLM Inference Framework. Mainstream Large Language Model (LLM) inference frameworks, such as vLLM [233] and LightLLM [234], do not support CPU inference. Recently, Intel introduced an LLM model named Q8chat [235], trained using their 4th Generation Intel Xeon® Scalable Processors. However, the inference code for Q8chat is not yet publicly available. To address this gap, we have developed our inference framework based on the open-source LightLLM framework [234] by replacing the backend with a CPU inference backend. Figure 4.9 illustrates our implementation. In our framework, the HTTPserver frontend receives LLM inference requests and forwards the tokenized requests to a router. The router is responsible for distributing these requests to different CPU backend instances. Each CPU backend instance is equipped with a Key-Value (KV) cache [236],

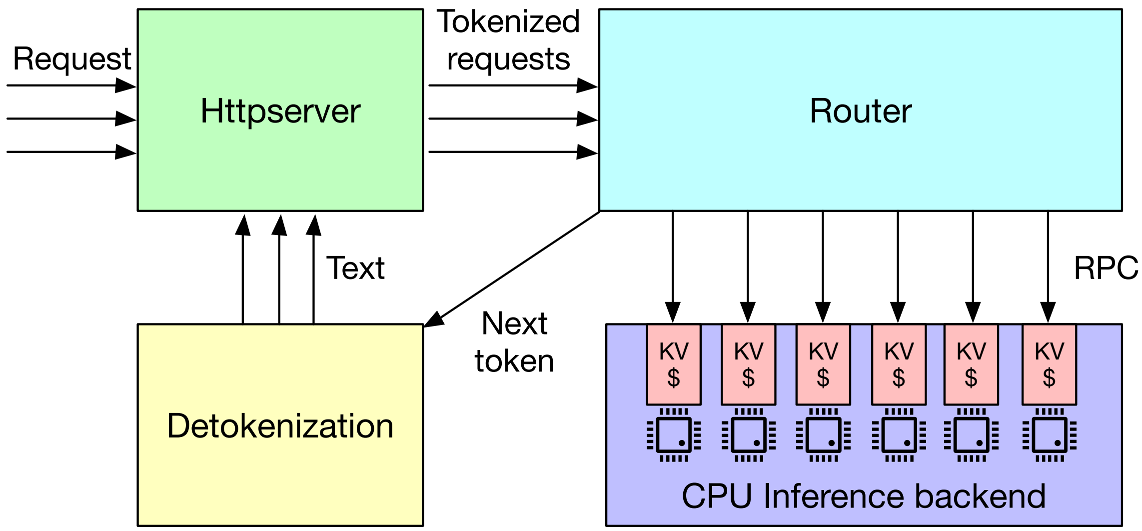
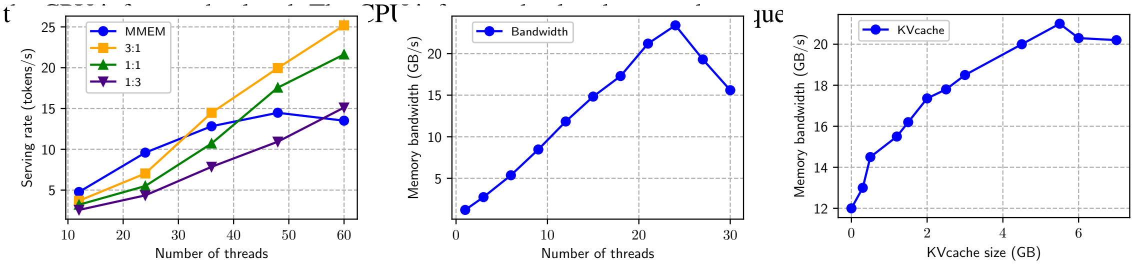


Fig. 4.9: **LLM inference framework.** The Httpserver receive requests and forward the tokenized requests to



(a) LLM inference serving rate vs. number of threads

(b) Memory bandwidth vs. number of threads for a single backend

(c) Memory bandwidth vs. KVcache size for a single backend

Fig. 4.10: **CPU LLM inference.**

a widely used technique in large language model inference. It's worth noting that KV caching, despite its name, differs from the traditional 'key-value store' in system architecture. KV caching occurs during multiple token generation steps, specifically within the decoder. During the decoding process, the model starts with a sequence of tokens, predicts the next token, appends it to the input, and repeats this generation process. This is how models like GPT [232] generate responses. The KV cache stores key and value projections used as intermediate data within this decoding process to avoid recomputation for each token generation. Prior research [236] has shown that KV caching is typically memory-bandwidth bound, as it is unique for each sequence in the batch, and different requests typically do not share the KV cache since the sequences are stored in separate contiguous memory spaces [237].

4.5.1 Methodology and Software Configurations

To investigate the benefits of CXL memory extension for applications with high memory bandwidth demands and limited MMEM bandwidth availability, we employ the SNC-4 configuration to divide a single CPU into four sub-NUMA nodes. Each node is equipped with two DDR5-4800 memory channels, facilitating an early memory bandwidth saturation of 67 GB/s (§4.3). We examine three distinct interleaving policies (3:1, 1:1, 1:3), detailed in Table 4.1. The CPU inference backend is configured with 12 CPU threads, and memory allocation is strictly bound to a single sub-NUMA domain. This domain includes two DDR5-4800 channels and a 256 GB A1000 CXL memory expansion module via PCIe. By binding allocations to a single node, we ensure the initial saturation of the DDR5 channels. Our experiments utilize the Alpaca 7B model [238], an advancement of the LLaMA 7B model, requiring 4.1GB of memory. The workload, derived from the LightLLM framework [234], includes a wide range of chat-oriented questions. A single-threaded client machine on a baseline server sends HTTP requests with various LLM queries to mimic real-world conditions. The client ensures continuous operation of the CPU inference backends by maintaining a constant stream of requests. The prompt context is set to 2048 bytes to guarantee a minimum inference response size. We progressively increase the CPU inference backend count to monitor the LLM inference serving rate (in tokens/s).

4.5.2 Analysis

Fig. 4.10(a) displays the inference serving rates across various memory configurations as the thread count, i.e., the number of CPU inference backends, increases. Initially, the serving rate improves almost linearly with available memory bandwidth. However, at 48 threads, MMEM bandwidth saturation limits the serving rate, whereas the interleaving configurations leverage additional CXL bandwidth for continued scaling. With a significant number of inference threads (60), an MMEM:CXL = 3:1 interleaving significantly surpasses the MMEM-only approach by 95%.

Interestingly, among the interleaving policies, configurations with a higher proportion of data in main memory demonstrate superior inference performance. Contrary to expectations, we observe that operating entirely on main memory is 14% less effective than a MMEM:CXL ratio of 1:3 beyond 64 threads. This outcome is notable given CXL's inherently higher latency and reduced memory bandwidth (§ 4.3). Fig. 4.10(b) charts the memory bandwidth utilization, as measured by the Intel Performance Counter Monitor (PCM) [213], with increasing CPU thread counts within a single CPU inference backend. Initially, bandwidth utilization grows linearly with thread count, plateauing at 24.2 GB/s for 24 threads. This trend allows us to estimate a bandwidth of approximately 63 GB/s at 60 threads, reaching 82% of the theoretical maximum. Our microbenchmark findings, as detailed in §4.3, indicate that this level of bandwidth utilization may lead to significant latency spikes. These results corroborate the hypothesis that bandwidth contention plays a crucial role in the observed performance degradation.

Bandwidth contention may stem from either loading the LLM model or accessing the KV cache. Adjusting the prompt context to infinity enables the LLM model to continuously generate new tokens for storage in the KV cache. Fig. 4.10(c) illustrates the correlation between KV cache size and memory bandwidth consumption. The initial memory bandwidth of approximately 12 GB/s originates from I/O threads loading the model from memory. When storing information for a larger sequence of tokens in the KV cache, memory usage initially increases linearly. However, bandwidth utilization stops increasing beyond roughly 21 GB/s.

4.5.3 Insights

Interestingly, existing tiered memory management in the kernel does not consider memory bandwidth contention. Considering a workload that uses high main memory bandwidth(e.g., 70%), existing page migration policy(§4.2) tends to move data from slower tiered-memory (CXL) into MMEM, supposing that there is still enough memory capacity. As more data is written into the main memory, the memory bandwidth will continue to increase (e.g., 90%). In this case, the access latency will grow exponentially, resulting in an actual slowdown of the workload. This scenario will not be uncommon, especially for memory-bandwidth-bound applications (e.g., LLM inference). Therefore, the definition of tiered memory requires rethinking.

4.6 Cost Implications

Our comprehensive analysis in prior sections (§4.4, §4.5) reveals that the adoption of CXL memory expansion offers substantial benefits for data center applications, including comparable performance

Parameter	Description
P_s	Throughput when (almost) entire working set is spilled to SSD on a server. Normalized to 1 in the cost model.
R_d	Relative throughput when the entire working set is in main memory on a server, normalized to P_s.
R_c	Relative throughput when the entire working set is in CXL memory on a server, normalized to P_s.
D	The MMEM capacity allocated to each server. For completeness only, not used in cost model.
C	The ratio of main memory to CXL capacity on a CXL server. E.g. 2 means the server has 2x MMEM capacity than CXL memory.
$N_{baseline}$	Number of servers in the baseline cluster.
N_{cxl}	Number of servers in the cluster with CXL memory to deliver the same performance as the baseline.
R_t	Relative TCO comparing a server equipped with CXL memory vs. baseline server. E.g. If a server with CXL memory costs 10% more than the baseline server, this parameter is 1.1.

Table 4.3: **Parameters of our Abstract Cost Model.**

with operational cost savings. However, a significant hurdle in embracing such innovative technology as CXL lies in determining its Return on Investment (ROI). Despite having access to detailed technical specifications and benchmark performance results, accurately forecasting the Total Cost of Ownership (TCO) savings remains challenging. The complexity of simulating benchmarks at production scale, compounded by the limited availability of CXL hardware, exacerbates this issue. Traditional cost models in prior work [199], which could offer such forecasts, demand extensive internal and sensitive information that is often inaccessible. To overcome this barrier, we propose an Abstract Cost Model designed to estimate TCO savings independently of internal or sensitive data. This model leverages a select set of metrics obtainable through microbenchmarks, alongside a handful of empirical values that are simpler to approximate or access, providing a viable means to evaluate the economic viability of CXL technology implementation.

We use a capacity-bound application (Spark SQL) as an example to demonstrate how we develop our Abstract Cost Model, but our methodology can be extended to other types of workloads as well. For Spark SQL applications, the additional capacity enabled by CXL memory reduces the amount of data spilled to SSD and results in higher performance (throughput). This means fewer servers will be needed to meet the same performance target.

Given that the workload maintains a relatively consistent memory footprint (the size of the active dataset) during execution, we can approximate the execution time of the workload by dividing it into three distinct segments: (1) The segment processed using data stored in MMEM, (2) The segment processed using data stored in CXL memory, and (3) The segment processed using data that has been offloaded to SSD storage.

We first make these measurements from microbenchmarks on a single server:

- Baseline performance (P_s): Measure the throughput when (almost) all working set is spilled to SSD. The absolute number is not used in our cost model. Instead, we then normalize it to 1 in our cost model.

- Relative performance when the entire working set is in MMEM (R_d): Using the same workload, we measure the throughput when the entire working set is in MMEM and normalize it to P_s to get the relative performance (i.e., how much faster compared to the baseline).

- Relative performance when the entire working set is in CXL memory (R_c): Using the same workload, we measure the throughput when the entire working set is in CXL memory, and normalize it to P_s to get the relative performance.

We then formulate our cost model using the parameters outlined in Table 5.1. For a working set size of W, the execution time of the baseline cluster could be approximated as the sum of two segments: 1) the segment that is executed with data in MMEM; 2) the segment that is executed with data spilled onto SSD.

$$T_{baseline} = \frac{N_{baseline}D}{R_d} + (W - N_{baseline}D)$$

The execution time of the cluster with CXL memory could be approximated in a similar way. It includes the segment that is executed with data in main memory, in CXL memory, and spilled to SSD respectively.

$$T_{cxl} = \frac{N_{cxl}D}{R_d} + \frac{N_{cxl}D}{CR_c} + (W - N_{cxl}D - \frac{N_{cxl}D}{C})$$

To meet the same performance target, $T_{baseline} = T_{cxl}$:

$$\frac{N_{baseline}D}{R_d} - N_{baseline}D = \frac{N_{cxl}D}{R_d} + \frac{N_{cxl}D}{CR_c} - N_{cxl}D - \frac{N_{cxl}D}{C}$$

With some simple transformations, we get the ratio between N_{cxl} and $N_{baseline}$:

$$\frac{N_{cxl}}{N_{baseline}} = \frac{CR_c(R_d - 1)}{R_cR_d(C+1) - CR_c - R_d}$$

TCO saving can then be formulated as follows.

$$TCO_{saving} = 1 - \frac{TCO_{cxl}}{TCO_{baseline}} = 1 - \frac{N_{cxl}R_t}{N_{baseline}}$$

For example, suppose $R_d = 10$, $R_c = 8$, $C = 2$, we get $\frac{N_{cxl}}{N_{baseline}} = 67.29\%$ from the cost model. This means that by using CXL memory, we may reduce the number of servers by 32.71%. And if we further assume $R_t = 1.1$ (a server with CXL memory costs 10% more than the baseline server), the TCO saving is estimated to be 25.98%.

Our Abstract Cost Model provides an easy and accessible way to estimate the benefit from using CXL memory, providing important guidance to the design of the next-generation infrastructure.

Extending Cost Model for more realistic scenarios. In line with previous research [199], our Abstract Cost Model is designed to be adaptable, allowing for the inclusion of additional practical infrastructure expenses such as the cost of CXL memory controllers, CXL switches (applicable in CXL 2.0/3.0 versions), PCBs, cables, etc., as fixed constants. However, a notable constraint of our current model is its focus on only one type of application at a time. This becomes a challenge when a data center provider seeks to evaluate cost savings for multiple distinct applications, each with unique characteristics, especially in environments where resources are shared (for instance, through CXL memory pools). This scenario introduces complexity and presents an intriguing challenge, which we acknowledge as an area for future investigation.

4.7 Related Work

4.8 Summary

Chapter 5

Conclusions and Future Work

YT: – working in progress –

In this dissertation, we take a top-down approach and explore the optimal memory management solutions for three important layers in the cloud stack, i.e. Service, OS and Hardware.

5.1 Future Work

We next discuss problems that we leave open in this dissertation and on-going work.

5.1.1 CXL-based KV Cache Storage

Autoregressive large language models (LLMs) generate output tokens sequentially, where the generation of each token involves the attention computation using key-value (KV) of its preceding tokens [239–241]. This sequential dependency makes LLM inference both compute- and memory-intensive. LLM inference typically includes two stages: the prefill stage, where all input tokens are processed to generate the initial output token, and the decode stage, where the rest of the output tokens are generated one by one until the model generates an end-of-sequence token [242–244].

For applications such as chatbot and coding assistant, LLM serving systems aim to minimize the time to finish the prefill stage, or time to first token (TTFT). In production, service-level objective (SLO) for TTFT is typically 400ms [243]. To meet such SLO, LLM serving systems often cache the previously-computed KV data of the preceding tokens (i.e., prefix) in GPU memory, to avoid re-computing them for future requests that have the same prefix [243, 245, 246]. Storing KV cache reduces the overall computational load and significantly improves throughput by trading memory for computation.

In production chatbot applications that support large context windows, the demand for KV cache storage grows rapidly by the number of inference requests from users, which cannot be fully accommodated by the limited and expensive GPU memory [247]. Researchers thus developed techniques to offload KV cache to CPU memory, leveraging the larger CPU memory capacity to reduce GPU memory pressure [246, 248, 249]. However, as larger LLMs and support for long-context inference requests continue to emerge, the approach of storing KV cache to CPU memory is still insufficient. For example, in LLaMA-2-7B, KV cache of token in FP32 precision is 1024KB; KV cache of a single request with 4096 tokens (maximum context length) is 4GB [250]. The memory demand from serving many concurrent long-context requests can easily overwhelm even high-end memory servers [245, 251].

Practitioners increasingly turn to more scalable memory architectures, such as Compute Express Link (CXL) memory [37,252,253], to address the growing memory demands of large-scale systems. CXL expands memory capacity by connecting additional DRAM to servers via PCIe, while maintaining low-latency access. It offers a promising solution to the KV cache storage demand in LLM serving.

In this paper, we propose to leverage CXL memory for storing KV cache, with the goal to improve serving throughput while retaining SLO on TTFT, and reduce KV cache storage pressure for the upper-level LLM serving system. This paper makes the following contributions:

- We present the first measurement of CXL-GPU interconnect and evaluate its feasibility for KV cache storage. We show that the data-transfer latency and bandwidth on CXL-GPU interconnect is on par with CPU-GPU interconnect.

- We present our design of CXL-based KV cache storage interface and evaluate its performance improvement to LLM serving, on our platform that is the first to successfully integrate ASIC-CXL device and GPU. Our results show competitive TTFT achieved by CXL-based prefix caching.

- We examine the cost-efficiency in using CXL for KV cache storage in production via Return on Investment (ROI) modeling. Estimates show a promising reduction in GPU compute cost when using CXL for KV cache storage. We also identify promising future research directions.

We now present the design and implementation of our CXL-based KV cache storage interface for LLM serving. We also describe the hardware platform used to evaluate our design.

Design and implementation. Our goal is to develop a CXL storage interface, named *KVExpress*, which can be integrated into existing LLM serving systems for saving and loading KV cache of inference requests. *KVExpress* provides two external APIs to its upper-level serving system: `save` and `load`. The `save` takes a unique identifier of a token chunk as input, and copies its KV cache from GPU to CXL memory. The `load` takes a unique identifier of a token chunk as input, and finds if its KV cache exists in CXL memory, if so, copies the KV cache from CXL memory to GPU. A token chunk can consist of one or more tokens. The unique identifier of a token chunk t_i for a sequence is the hash of the content of t_i and the hash of its prefix $\langle t_0, ..., t_{i-1} \rangle$. If the prefix of a sequence of a current request has been computed and saved into CXL, *KVExpress* will load the KV cache of the prefix from CXL and use it when computing for this request [245].

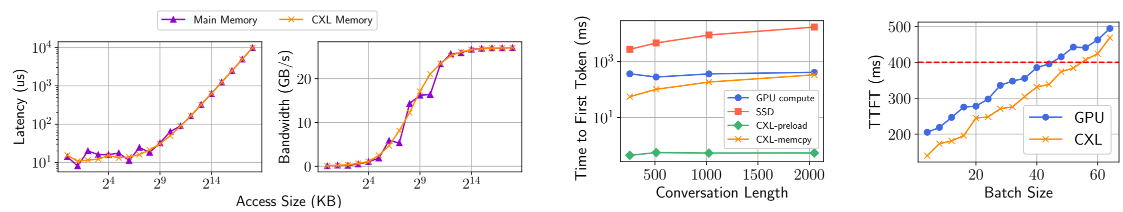
To avoid calling `save` and `load` too frequently and incurring unnecessary overhead to the upper-level serving system, `save` is called only when a request is finished so the KV cache of all the tokens for that request is saved at once; `load` is called for a request prior to its prefill computation.

We implement our design of *KVExpress* in gpt-fast [254], a simple and low-latency text generation system with support on a number of widely-used inference optimizations [255–257] and open-source LLMs [250, 258]. We further modify gpt-fast to support our evaluation on batched inference.

Hardware platform. Our single socket server is equipped with Intel Xeon Platinum processors [259], 1TB of 4800 MHz DDR5 memory, an NVIDIA H100 GPU with 96GB HBM, and a CXL memory expansion card with 256 GB of DDR5 memory at 4800 MHz [253]. While prior works [260–262] have explored utilizing CXL for accelerators, to our knowledge, our work is the first implementation to successfully integrate a real ASIC-CXL device and a GPU within a single inference server.

5.2 Performance Evaluation

In Section 5.2.1, we measure the latency and bandwidth of CXL-GPU interconnect for data transfer to assess the feasibility of storing KV cache on CXL devices. In Section 5.2.2, we compare the TTFT of KV re-compute, prefix caching with CXL, and prefix caching with GPU, to understand if *KVExpress* can achieve similar TTFT as existing approaches for prefill requests under varying context lengths. In Section 5.2.3, we study the maximum batch size achieved while retaining a given SLO on TTFT between KV re-compute and prefix caching with CXL.



(a) CPU-GPU/CXL-GPU interconnect. (b) TTFT comparison. (c) Max BS under SLO.

Fig. 5.1: **Experiment results.** (a) Latency and bandwidth measurements across different access sizes, CXL-GPU interconnect performs similarly as CPU-GPU interconnect. (b) TTFT comparison between KV re-compute and prefix caching with CXL or GPU. (c) Serving throughput comparison under a fixed SLO constraint (400ms).

5.2.1 Measurements on CXL-GPU interconnect performance

KV cache storage requires low-latency access (e.g., from host memory to GPU memory). Although prior studies [252,253] show that accessing CXL memory from the host CPU is over $2\times$ slower than accessing local memory, none of their measurements involves any interaction with the GPU. In this paper, we evaluate the performance characteristics of the CXL-GPU interconnect by measuring the latency and bandwidth of copying data from CXL memory to the GPU. Transferring in the reverse direction yields similar results [263]. Since CXL memory devices are exposed to the system as NUMA nodes without CPUs by default [253], we allocate a set of host buffers on the CXL NUMA node and use `cudaMemcpyAsync` to copy data between the host buffers and GPU device buffers allocated via the CUDA API [264]. We evaluated transferring data of sizes ranging from 1KB to 256MB.

Figure 5.1(a) shows our experiment results: the performance of the CXL-GPU interconnect **is unexpectedly on par with** traditional CPU-GPU memory transfers, exhibiting no significant slowdown. Latency remains low for smaller access sizes but increases exponentially once the size exceeds 64KB. Meanwhile, bandwidth increases almost linearly with data size and saturates around 4MB. This indicates that, while the CPU oversees the data transfer, the data path actually bypasses the host's local memory, flowing directly from CXL memory to GPU buffers via PCIe. Our results demonstrate that the CXL-GPU interconnect operates efficiently with minimal latency overhead, positioning it as a promising expansion for KV cache storage in addition to CPU memory.

5.2.2 Evaluation on TTFT under varying input context length

Given that CXL-GPU interconnect performs nearly the same as CPU-GPU interconnect, we further study if CXL-based KV cache storage can achieve similar TTFT as existing approaches in completing the prefill stage computation for an inference request. We evaluate three approaches:

- **KV re-compute:** Compute KV data of all input tokens for the request with GPU.
- **Prefix caching with CXL:** Load KV cache of the prefix tokens for the request from CXL to GPU.
- **Prefix caching with GPU:** Store and use KV cache in GPU for the prefix tokens for the request.

We measure the TTFT of the aforementioned approaches on conversation requests of input length ranging from 256 to 2048 tokens from the ShareGPT-Vicuna-Unfiltered dataset [265]. We use the LLaMA-2-13B as the underlying model for our evaluation. Figure 5.1(b) shows the TTFT (y-axis in log-scale) achieved by the evaluated approaches for requests of varying input context length (x-axis).

Compared to the other approaches, prefix caching with GPU (denoted as "PC-GPU" in Figure 5.1(b)) achieves the smallest TTFT (0.44ms to 0.56ms) constantly across different input context lengths. Such performance is expected as there is no data transfer latency and computation of KV data is only needed for tokens after the prefix. This approach is an optimal baseline that is however difficult to achieve in practice due to limited memory capacity of existing GPU models and the rapidly growing demand of KV cache storage in LLM serving.

Comparing prefix caching with CXL (denoted as "PC-CXL") and KV re-compute, prefix caching with CXL performs at least as good as computing KV data on GPU from scratch. Prefix caching with CXL achieve TTFT ranging from 55ms to 336ms, with slight increase in latency as input size length grows. The close performance gap between storing prefix KV cache in CXL memory and full KV re-computation indicates that there is a potential opportunity to reduce GPU compute cost with adaptation of CXL devices for memory capacity expansion in LLM inference.

5.2.3 Evaluation on serving throughput while adhering SLO

By storing the KV cache of the inference request prefix in CXL memory and thus reducing re-computation during the prefill stage, we can effectively reduce the computational load on the GPU. The saved GPU compute can be re-allocated to handle a larger number of concurrent inference requests. In other words, the LLM serving system can achieve a higher serving throughput, by handling a larger batch size of inference requests using the saved GPU compute, while maintaining the same SLO on TTFT [243].

Figure 5.1(c) shows the TTFT achieved by KV re-compute and prefix caching with CXL under varying batch size. The horizontal red-dashed line indicates our SLO limit–the maximum TTFT that can be tolerant in production. The typical SLO is 400ms used for LLaMA-2 [266]. As shown in Figure 5.1(c), with KV re-compute, the evaluated serving system (§**??**) can handle a maximum batch size of 44 before hitting the SLO limit. On the other hand, when leveraging CXL for storing KV cache, the system can handle a maximum batch size of 57, which is a **30% increase** compared to KV re-compute. Our initial evaluation on SLO-adhering serving throughput highlights the performance benefits of utilizing CXL memory for KV cache storage, particularly in scenarios that require efficient scaling under strict latency requirements.

5.3 Cost-Efficiency Modeling

We develop a model to estimate the Return on Investment (ROI) of deploying *KVExpress* in production. Conceptually, each prefill request consists of two distinct parts: 1) Loading KV cache data for the prefix (i.e., the history context) from CXL memory; 2) Performing computation on the new prompt (i.e., the follow-up prompt in multi-round conversations).

By replacing computation with memory accesses, we reduce the overall computational load, thereby lowering the demand for FLOP/s while still meeting the same SLO. This results in signifi-

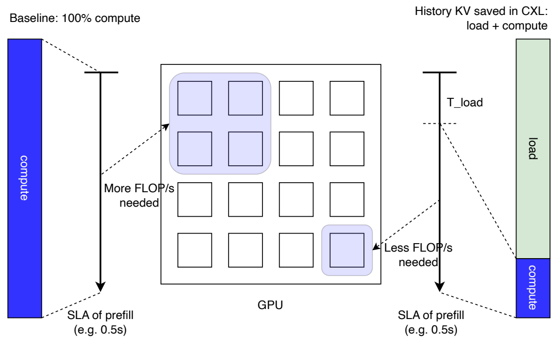


Fig. 5.2: Example of ROI modeling: replace computation with memory access

cant cost savings for LLM inference (Figure 5.2):

- **Assumption:** Assume a GPU has a computational power of 100 TFLOP/s, an average prefill request requires 25 TFLOP of computation, and the SLO for prefill is 0.5 seconds.
- **Baseline:** To complete the prefill request within the SLO, each request demands 50 TFLOP/s (25 TFLOP/0.5s), meaning a single GPU can serve 2 prefill requests.
- **KVExpress:** By spending 0.1s loading KV cache data of the history context, we reduce the computational demand to 2.5 TFLOP (assuming the new prompt accounts for 10%). To meet the same SLO, the remaining computation must be finished within 0.4s, requiring 6.25 TFLOP/s (2.5 TFLOP/0.4s). In this case, a single GPU can serve 16 prefill requests, yielding an **8x improvement over the baseline**.

This allows for a reduction of 87.5% in the number of GPUs required for the same prefill SLO, resulting in substantial cost savings for LLM inference applications (more details in Appendix [**?**]).

5.4 The Potential of Memory Disaggregation for AI/ML Workloads

Storing KV cache in GPU memory for LLM inference can quickly lead to memory saturation, limiting serving scalability and performance. KV cache storage on CPU memory becomes limited as model size and request context length increase. To that extent, we explore CXL memory for KV cache offloading, in which CXL offers expanded capacity with low-latency access. Our preliminary results show that CXL-CPU data transfer has similar latency and bandwidth as the CPU-GPU counterpart. In addition, CXL-based KV cache offloading provides similar performance compared to full

Table 5.1: ROI Modeling

C_0	Avg. FLOPs needed by a prefill request in an initial request. Can be estimated as $C_0 = 2ML$, where M is the model parameters and L is the avg. sequence length.
C_1	FLOPs needed by new prompt in a follow-up request. Can be estimated as $C_1 = rC_0$, where r is the avg. ratio of the new prompt (e.g., 10%).
T_{slo}	SLO of prefill (e.g., 0.5s).
T_{load}	Avg. time to load KV cache from memory (e.g., 0.1s).
P	Computation power (FLOP/s) of the GPU.
P_0	FLOP/s needed for the initial request. $P_0 = C_0/T_{slo}$
P_1	FLOP/s needed for the new prompt. $P_1 = C_1/(T_{slo} - T_{load})$
R_{gpu}	Request per second (RPS) a single GPU can support. $R_{gpu} = P/(P_0(1-h) + P_1 h)$, where h is the ratio of multi-round requests.
N_{cxl}	Number of GPUs needed using our CXL memory scheme. $N_{cxl} = \lceil R/R_{gpu} \rceil = \lceil \frac{R}{P/(P_0(1-h)+P_1 h)} \rceil$
$N_{baseline}$	Number of GPUs needed without any KV cache stored (i.e., all data discarded after prefill). $N_{baseline} = \lceil \frac{R}{P/P_0} \rceil$

KV re-compute on GPUs, while supporting larger workloads. Specifically, using CXL memory for KV cache storage increased the maximum batch size by 30%, while maintaining the same SLO on TTFT. Our cost-efficiency analysis further shows the potential for using CXL memory to substantially reduce the GPU compute cost for high-throughput LLM serving under SLO. Looking ahead, future work will explore the integration of CXL memory with multi-GPU systems, focusing on maintaining cache coherence across GPUs that could further enhance the scalability and efficiency of LLM inference.

Appendix A

Appendix

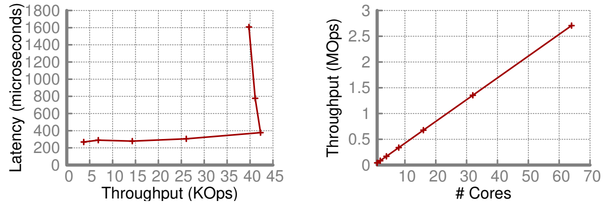
A.1 Jiffy: Additional Evaluation

We now present additional results for Jiffy, including: an evaluation of its control plane (Appendix A.1.1), and sensitivity analysis for various system parameters (Appendix C.3).

A.1.1 Controller Performance

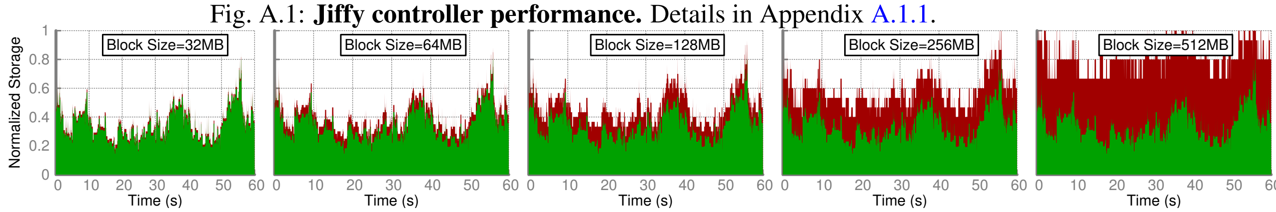
Jiffy adds several components at the controller compared to Pocket, including all of metadata management, lease management and handling requests for data repartitioning. As such, we expect its performance to be lower than Pocket's metadata server. We deem this to be acceptable as long as it can still handle control plane request rates typically seen for real world workload, *e.g.*, a peak of a few hundred requests per second, including lease renewal requests, for all of our evaluated workloads and those evaluated in [42].

Figure A.1(a) shows the throughput-vs-latency curve for Jiffy controller operations on a single CPU core of an m4.16xlarge EC2 instance. The controller throughput saturates at roughly 42 KOps, with a latency of 370us. While this throughput is lower than Pocket ($\sim$ 90KOps per core), it is more than sufficient to handle control plane load for real-world workloads. In addition, the throughput scales almost linearly with the number of cores, since each core can handle requests independent of other cores for a distinct subset of virtual address hierarchies (Figure A.1(b)). Moreover, the Jiffy control plane readily scales to multiple servers by partitioning the set of virtual address hierarchies across them.

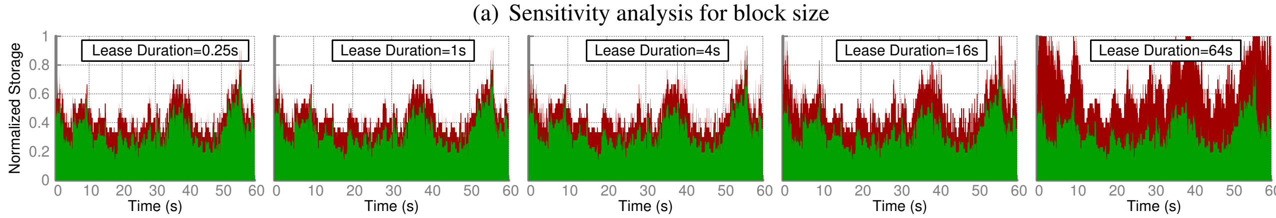


(a) Controller throughput vs. latency on a single CPU core.
(b) Controller throughput scaling with multiple cores.

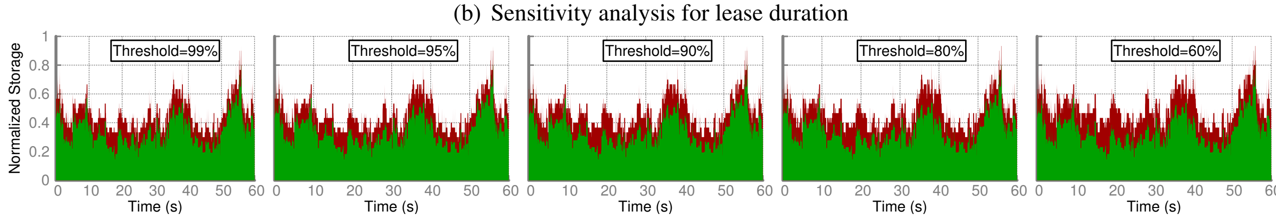
Fig. A.1: **Jiffy controller performance.** Details in Appendix A.1.1.



(a) Sensitivity analysis for block size



(b) Sensitivity analysis for lease duration



(c) Sensitivity analysis for (high) repartition threshold

Fig. A.2: **Jiffy sensitivity analysis** for (a) block size (b) lease duration and (c) repartition threshold for the file data structure. Green area corresponds to used capacity, while red area corresponds to allocated capacity under Jiffy. See Appendix C.3 for details.

A.1.2 Sensitivity Analysis

We now perform sensitivity analysis for various system parameters in Jiffy, including block size (§2.1.1), lease duration (§2.1.2) and thresholds for data repartitioning (§2.1.3). We use files as our underlying data structure, and use the Snowflake workload from Figure 2.1. These results can be contrasted directly with Figure 2.10(a) (center), which corresponds to our default system parameters (128MB blocks, 1s lease duration and 95% of block occupancy as repartition threshold). For each parameter that we vary, the other to remain fixed at their default values.

Block size (Figure A.2(a)). As discussed in §2.1.1, the block size in Jiffy exposes a tradeoff between the amount of metadata that needs to be stored at the control plane, and resource utilization.

This is confirmed in Figure A.2(a), where increasing the block size from 32MB to 512MB increases the disparity between allocated and used capacity, and therefore decreases the resource utilization. The default block size in Jiffy is set to 128MB for two main reasons: (1) it allows high enough utilization with low enough metadata overhead (a few megabytes for even thousands of gigabytes of application data), and (2) it is the default block size used in existing data analytics platforms; as such, 128MB blocks ensure seamless compatibility with such frameworks.

Lease duration (Figure A.2(b)). As shown in Figure A.2(b), lease duration in Jiffy controls resource utilization over time. As we increase lease durations from 0.25 seconds to 64 seconds, resource utilization increases since Jiffy does not reclaim (potentially unused) resource resources from jobs until their leases expire. At the same time, if we keep lease duration too low, applications would renew leases too often, resulting in higher traffic to the Jiffy controller. We find a lease duration of 1s to be a sweet spot, ensuring high enough resource utilization, while ensuring the number of lease requests for even thousands of concurrent applications is only a few thousand requests per second — well within Jiffy controller's limits on a single CPU core.

Repartition threshold (Figure A.2(c)). Finally, Figure A.2(c) shows the impact of (high) repartition threshold on resource utilization. As expected, lowering the repartition threshold leads to poor utilization, since it triggers pre-mature allocation of new blocks to most files in our evaluated workload. However, since the size of the block (128MB) is much smaller than the amount of data written to each file in the workload (often several gigabytes), this overhead is relatively small when compared to effect of other parameters. However, a larger value of high repartitioning threshold results in more frequent block allocation requests to the controller; we find that our default value of 95% provides a reasonable compromise between resource utilization and number of control plane requests.

Supplementary Materials

A. Multiplexing $M + N$ Iterator Executions for Maximizing Pipeline Utilization

We claimed in §3.3.3 that if $t_c = \eta \cdot t_d$ for all offloaded iterator executions, it is always possible to multiplex $m + n$ concurrent iterator executions and fully utilize all memory and logic pipelines. We prove our claim by providing a staggered scheduling algorithm (Algorithm 1) that ensures such multiplexing across $m + n$ iterator executions. The scheduler processes $m + n$ iterator execution requests, assigning each a memory pipeline, a logic pipeline, and staggered start times. These requests are then executed in the respective memory pipelines. Through this staggered scheduling approach, Jiffy fully utilizes the n memory pipelines and m logic pipelines, ensuring no resources are wasted. Note that this algorithm is a simplified version to illustrate the potential for full pipeline saturation under the given condition. Jiffy's scheduler implements a real-time algorithm to multiplex incoming requests on the fly.

Algorithm 1 Staggered-Scheduling

$m, n \leftarrow$ number of logic, memory pipelines
$L_i, M_j \leftarrow i^{th}$ logic pipeline, j^{th} memory pipeline
$t_d \leftarrow$ data fetch time per pointer traversal iteration
while true **do**
    Dequeue $n + m$ requests from network stack
    for $i \leftarrow 1$ **to** $m + n$ **do**
        Assign request R_i to $(M_{i \bmod n}, L_{i \bmod m})$
        Schedule R_i to start at time $(i-1) \cdot \frac{t_d}{n}$
    Start requests as scheduled at memory pipelines

A.1 PULSE Empirical Analysis

Prior studies have shown that real-world data-centric cloud applications spend a significant fraction of time traversing pointers, as summarized in Fig. A.1.

B. PULSE Supported Data Structures

We adapt 13 data structures across 4 popular open-sourced libraries to PULSE's iterator abstraction (§??). In particular, we outline how the data structure implementations for certain operations can be expressed using `init()`, `next()`, and `end()`. For simplicity and readability, (i) we assume that the data structure developer defines a macro, `SP_PTR_(variable_name)`, as the address of the variable resides on the `scratch_pad`, and (ii) we omit obvious type conversions for de-referenced pointers.

We analyze two widely used categories of data structures: lists and trees. In our analysis, we find that the top-level data structure APIs (i.e., the APIs used by applications) use the same base function under the hood. For instance, list and forward list in the STL library share the same internal function, `std::find()`. We summarize our findings in Table A.1, including the data structure libraries, their category, the top-level data structure APIs, and the internal base function.

List structures. Our surveyed list structures already follow the execution flow of PULSE iterator: `init()`, `next()`, and `end()`.

These data structures generally have compute-intensive `end()` functions to check multiple termination conditions, while their `next()` function simply dereferences a single pointer to the next node. Listing A.1 and Listing A.2 demonstrate a linked list with two termination conditions: (i) value is found or (ii) search reaches the end. To indicate which condition is met, a special flag (*e.g.*, `KEY_NOT_FOUND`) is written on the `scratch_pad`. Listing A.3 and Listing A.4 describe a bitmap that uses a hashtable internally, where colliding entries are stored in linked lists within the same bucket. As such, the PULSE iterator interface resembles that of `std::list` quite closely.

Tree-like data structures. Compared to list structures, tree data structures require more computation in the `next()` function, as the next pointer is determined based on the value in the child node.

Application	% of time spent in pointer traversal
GraphChi [94]	$\sim 93\%$
MonetDB [102]	$70\% - 97\%$
GC in Spark [67]	$\sim 72\%$
VoltDB [103]	Up to 49.55%
MemC3 [104]	Up to 21.15%
DBx1000 [105]	$\sim 9\%$
Memcached [106]	$\sim 7\%$

(a) Survey from prior studies

Fig. A.1: **Time cloud applications spend in pointer traversals** based on prior studies

For instance, in `Btree` (Listing A.5, A.6), the next function iterates through internal node keys, comparing them to the search key. Interestingly, `std::map` (Listing A.7, A.8) and Boost AVL trees (Listing A.9, A.10) share the same offload function structure, with only minor implementation and naming differences.

Table A.1: Additional data structure supported by PULSE.

Data Structure	**Category**	**Library**	**Data structure API**	**Internal function**	**Original code**	**PULSE code**
List	List	STL	`std::find(start, end, value)`	`std::find(start, end, value)`	Listing A.1	Listing A.2
Forward list						
Bimap		Boost	`find(key, hash)`	`find(key, hash)`	Listing A.3	Listing A.4
Unordered map						
Unordered set						
Btree	Tree	Google	`find(&key)`	`internal_locate_plain_compare(key, iter)`	Listing A.5	Listing A.6
Map		STL		`_M_lower_bound(x, y, key)`	Listing A.7	Listing A.8
Set						
Multimap						
Multiset						
AVL tree		Boost		`lower_bound_loop(x, y, key)`	Listing A.9	Listing A.10
Splay tree						
Scapegoat tree						

B.1 List data structure in STL library

Listing A.1: C++ STL realization for `std::find()`

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

node* find(node* first, node* last,
     const value_type& value)
{
    for (; first != last;
         first=first->next)
        if (first->value == value)
            return first;
    return last;
}
```

Listing A.2: PULSE realization for `std::find()`

```
class list_find : chase_iterator {

    init(void *value, void* first) {
        *SP_PTR_VALUE = value;
        cur_ptr = first;
    }

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

    bool end() {
        if (*SP_PTR_VALUE ==
             cur_ptr->value) {
            *SP_PTR_RETURN = cur_ptr;
            return true;
        }
        if (cur_ptr->next == NULL) {
            *SP_PTR_RETURN =
                KEY_NOT_FOUND;
            return true;
        }
```

B.2 List data structure in Boost library

~~**Listing A.3:** Boost realization for `bimap::find`~~()

```
struct node {
    key_type key;
    struct node* next;
    value_type value;
};
void* find(const key_type& key, const hash_type& hash) const
{
    // The bucket start pointer can be pre-computed before offloading
    std::size_t buc = buckets.position(hash(key));
    node_ptr start = buckets.at(buc)
    for(node_ptr x = start; x != NULL; x = x->next){
        if(key == x->key){
            return x;
        }
    }
    return NULL;
}
```

~~**Listing A.4:** PULSE realization for `bimap::find`~~()

```
class bimap_find : chase_iterator {
public:
    key_type key;

    init(void *key, void* start) {
        *SP_PTR_KEY = key;
        cur_ptr = start;
    }

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

    bool end() {
```

B.3 Tree data structure in Google library

Listing A.5: Google realization for btree::internal_locate_plain_compare()

```
#define kNodeValues 8
struct btree_node {
    bool is_leaf;
    int num_keys;
    key_type keys[kNodeValues];
    btree_node* child[kNodeValues + 1];
};
IterType
    btree::internal_locate_plain_compare(const
    key_type &key, IterType iter) const
    {
    for (;;) {
        int i;
        for(int i = 0; i <
            iter->num_keys; i++) {
            if(key <= iter->keys[i]) {
                break;
            }
        }
        if (iter.node->is_leaf) {
            break;
        }
        iter.node = iter.node->child(i);
    }
    return iter;
}
```

Listing A.6: PULSE realization for btree::internal_locate_plain_compare()

```
class btree_find_unique :
    chase_iterator {
    init(void *key, void* iter) {
        *SP_PTR_KEY = key;
        cur_ptr = iter;
    }

```

B.4 Tree data structure in STL library

Listing A.7: C++ STL realization for `map::find()`

```
struct node {
    key_type key;
    node* left;
    node* right;
};

_M_lower_bound(node* x, node* y, const
     key_type& key)
{
    while (x != 0) {
        if (x->key <= key) {
            y = x;
            x = x->left;
        } else {
            x = x->right;
        }
    }
    return y;
}
```

Listing A.8: PULSE realization for `map::find()`

```
class map_find : chase_iterator {
    init(void *key, void* x, void* y) {
        *SP_PTR_KEY = key;
        *SP_PTR_Y = y;
        cur_ptr = x;
    }

    void* next() {
        if (cur_ptr->key <= *SP_PTR_KEY) {
            *SP_PTR_Y = cur_ptr;
            cur_ptr = cur_ptr->left;
        } else {
            cur_ptr= cur_ptr->right;
        }
        return cur_ptr->left;
    }
```

B.5 Tree data structure in Boost library

Listing A.9: Boost realization for avltree::find()

```
static node_ptr lower_bound_loop
(node_ptr x, node_ptr y, const KeyType &key)
{
    while(x){
        if(x->key >= key)) {
            x = x->right;
        }
        else{
            y = x;
            x = x->left;
        }
    }
    return y;
}
```

Listing A.10: PULSE realization for avltree::find()

```
class avltree_find : chase_iterator {
public:
    key_type key;
    void* y;

    init(void *key, void* x, void* y) {
        *SP_PTR_KEY = key;
        *SP_PTR_Y = y;
        cur_ptr = x;
    }

    void* next() {
        if(cur_ptr->key >= *SP_PTR_KEY) {
            cur_ptr = cur_ptr->right;
        }
        else{
            *SP_PTR_Y = cur_ptr;
            cur_ptr = cur_ptr->left;
```

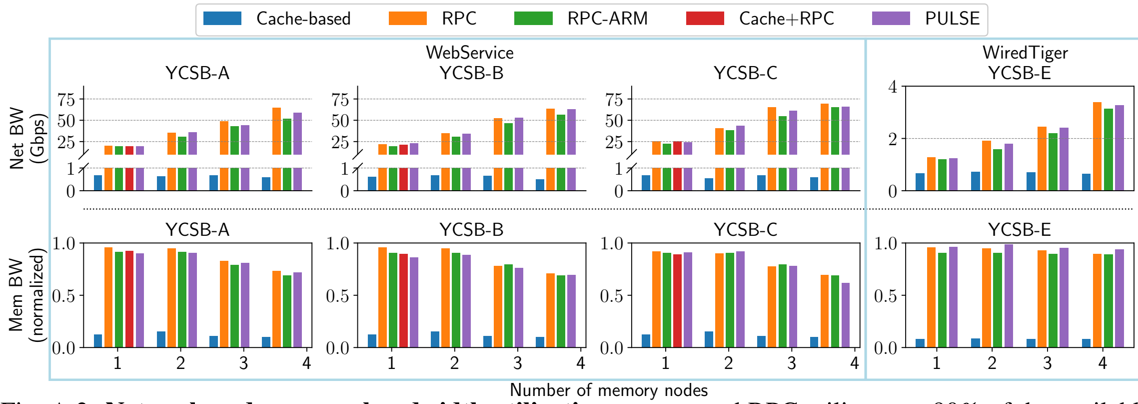


Fig. A.2: **Network and memory bandwidth utilization.** PULSE and RPC utilize over 90% of the available memory bandwidth, while the cache-based approach suffers from swap system overhead. In Webservice, the network bandwidth becomes the bottleneck due to large 8 KB data transfers.

C. PULSE Additional Evaluation Results

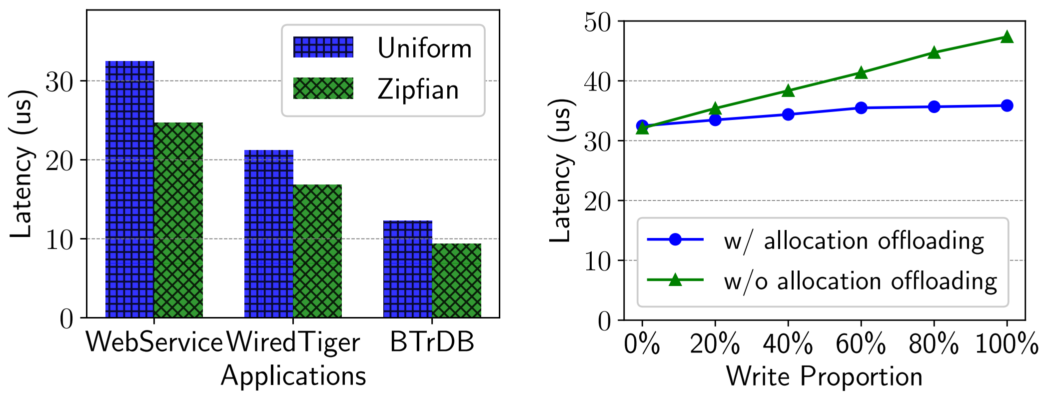
In this section, we provide additional evaluation results for PULSE.

C.1 Traditional Core Architecture vs. PULSE

We evaluate the impact of the PULSE architectural design (§**??**) by comparing PULSE against PULSE-CORE, an in-order processor built on PULSE's components. We denote C_x as in tightly-coupled core architecture, where x is the number of cores. We denote P_x_y as PULSE architecture, where x is the number of logic pipelines and y is the number of memory pipelines. Table A.2 shows the power, performance, and area usage of various configurations. The performance metrics are obtained by executing the WebService application with various configurations. In PULSE's disaggregated architecture, when the number of logic and memory pipelines is equal to that of a traditional core architecture, power and area usage are higher due to additional logic and buffering in the interconnect and scheduler. However, due to the nature of pointer traversal operations (§**??**), PULSE requires fewer logic pipelines to achieve similar performance. For example, to fully saturate the memory bandwidth of a single node, PULSE uses only one logic pipeline and four memory pipelines, while a traditional core architecture requires four cores. As a result, PULSE saves 20.12% in power with only a 7.2% latency overhead, primarily due to the additional scheduler and data movement between workspaces.

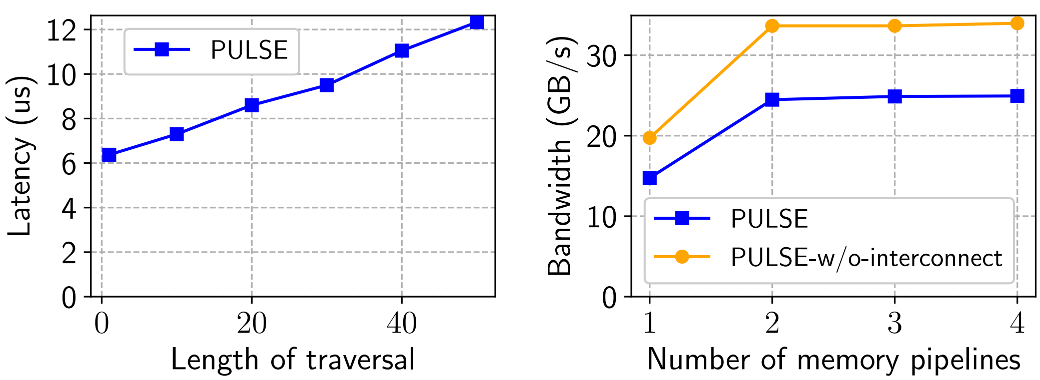
Config	Pwr (W)	LUT %	BRAM %	Tpt (Mops/s)	Lat (us)
C_1	67.76	14.73	14.57	0.41	33.25
C_2	75.47	20.46	18.73	0.63	33.73
C_3	84.57	28.66	31.83	0.87	34.66
C_4	89.77	37.10	34.17	1.20	35.11
P_1_1	56.74	11.76	16.34	0.51	37.57
P_1_2	59.47	14.87	18.38	0.73	36.74
P_1_3	64.78	16.64	22.37	1.01	38.46
P_1_4	72.47	18.37	25.84	1.24	38.37
P_2_1	67.37	17.73	20.37	0.48	40.27
P_2_2	77.37	21.38	22.38	0.76	39.47
P_2_3	81.21	26.22	26.76	0.99	41.37
P_3_3	86.15	37.21	30.12	1.03	40.98
P_2_4	83.21	30.13	31.21	1.19	40.37
P_4_4	95.64	46.42	39.84	1.21	41.47

Table A.2: Comparison between traditional core architecture and PULSE architecture.



(a) Impact of access pattern (b) Impact of modifications

Fig. A.3: (a) PULSE latency is up to $1.3\times$ lower for skewed than uniform access patterns due to caching. (b) Offloaded allocations in PULSE improve the WebService request latencies as the proportion of writes increases by reducing the number of round trips per allocation.



(a) Traversal length (b) Number of memory pipelines

Fig. A.4: **Sensitivity to traversal length and the number of memory pipelines.** (a) PULSE latency scales linearly with the length of traversal. (b) PULSE accelerator can saturate memory bandwidth with just two PULSE memory pipelines.

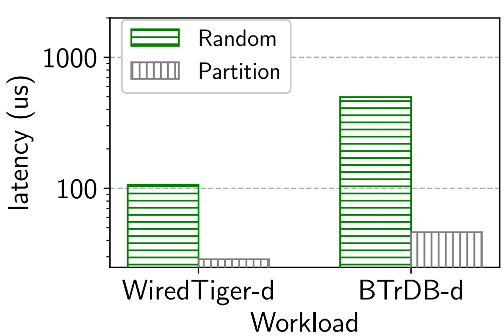
C.2 Network and Memory Bandwidth Utilization

We evaluate the network and memory bandwidth utilization of the three applications in Fig. A.2. For WiredTiger, PULSE and RPC utilize over 90% of the available memory bandwidth, while the Cache-based approach suffers from low network bandwidth and memory utilization due to swap system overhead. For WebService, the large 8 KB data transfers saturate the maximum bandwidth that the DPDK stack can sustain [125]. As a result, network bandwidth becomes the bottleneck, reducing PULSE and RPC memory bandwidth utilization under 3 and 4 memory nodes. The memory bandwidth is normalized, where 1.0 corresponds to 25 GB/s per node.

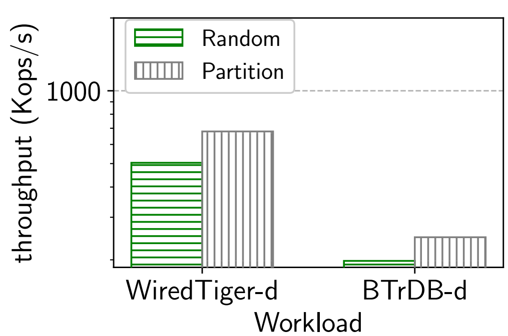
C.3 PULSE Sensitivity Analysis

We evaluate PULSE's sensitivity to workload characteristics and system parameters: access pattern, data structure modifications, traversal length, allocation policy, and the number of PULSE memory pipelines.

Impact of access pattern. While our evaluation so far has been confined to Zipfian workloads, we evaluate the impact of skewed access patterns on PULSE performance for all three applications. Our setup comprises a single 32GB memory node with a 2GB CPU node cache. Figure A.3(a) shows caching at the CPU node reduces the number of iterator requests offloaded to the PULSE accelerator for the skewed (Zipfian) workload, improving PULSE performance for such workloads by up to $1.33\times$ relative to uniform ones.



(a) Latency



(b) Throughput

Fig. A.5: **Allocation policy.** PULSE performs better with the partitioned allocation since it minimizes cross-node traversals.

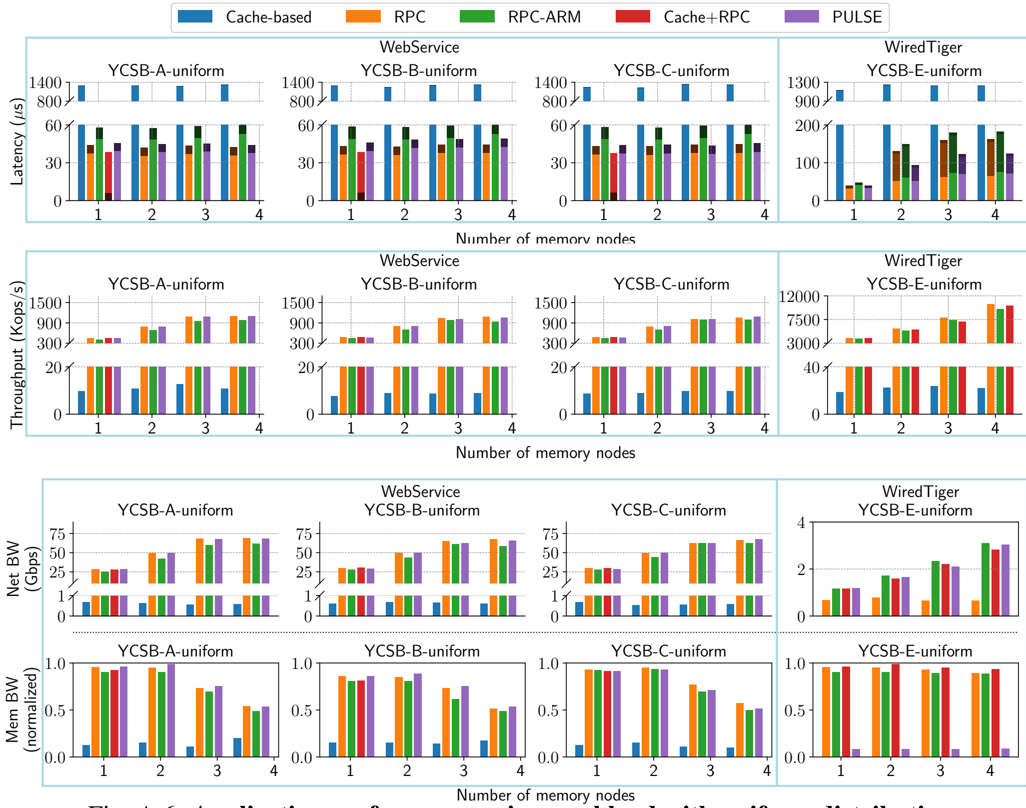


Fig. A.6: **Application performance using workload with uniform distribution.**

Impact of data structure modifications. Operations that modify data structures can require new memory allocations during traversal. Instead of returning control to the CPU node for allocations, PULSE populates the scratchpad for every request with a fixed number of pre-allocated memory regions. When a new allocation is initiated at the PULSE accelerator, it uses a pre-allocated memory region on the scratchpad. If all such regions (16 in our implementation) are used up in a single request, the traversal interrupts and returns to the CPU node. PULSE periodically replenishes pre-allocated entries, ensuring that allocation-triggered traversal interruptions are rare.

We evaluate the impact of data structure modifications in PULSE (§**??**) by increasing the proportion of writes for the WebService application on a single memory node. Figure A.3(b) shows that as the proportion of writes increases, PULSE without offloaded allocations experiences higher latencies (up to $1.4\times$) since each new node allocation requires two additional round trips; offloaded allocations reduce the allocation overhead to $< 1.1\%$.

Length of traversal. For simplicity, we evaluate traversal queries on a single linked list with varying numbers of nodes traversed per query. As expected, Fig. A.4(a) shows that the end-to-end execution latency for a linked list traversal scales linearly with the number of nodes traversed.

Allocation policy. We find that the allocation policy used for a data structure has a significant impact on application performance specifically for distributed traversals (Figs. A.5(a) and A.5(b)). We evaluated the WiredTiger and BTrDB workloads (that employ B+-Tree as their underlying data structure) with two allocation policies: one that partitions allocations in a way that ensures all nodes in half the subtree are placed on one memory node and the other half on another, and another that allocates memory uniformly across the two nodes (as in `glibc` allocator). The average latency for random allocations is $3.7-10.8\times$ higher than partitioned allocation since it incurs significantly more cross-node traversals. This shows that while uniformly distributed allocations can enable better system-wide resource utilization, it may be preferable to exploit application-specific partitioned allocations for workloads where performance is the primary concern.

Number of PULSE memory pipelines. We evaluate the number of PULSE memory access pipelines required to saturate PULSE's memory bandwidth on a single memory node. We used the same linked list as our traversal-length experiment due to its relatively low η value ($\sim$0.06), which allows us to stress the memory access pipeline without saturating the logic pipeline. Fig. A.4(b) shows that just 2 memory pipelines can saturate PULSE's the per-node memory bandwidth of 25 GB/s. We note that our 25 GB/s limit does not match the hardware-specified memory channel bandwidths; this is primarily due to our use of the vendor-supplied memory interconnect IP, required to connect all memory pipelines to all memory channels. Indeed, if we remove the IP and measure memory bandwidth when each memory pipeline is connected to a dedicated memory channel, PULSE can achieve a memory bandwidth up to 34 GB/s (shown as PULSE w/o Interconnect in Fig. A.4(b)).

PULSE performance with uniform workload. As illustrated in Fig. A.6, while sharing a similar trend as Zipfian distribution, all approaches experience higher latency compared to Zipfian distribution due to the ineffectiveness of caching. PULSE provides lower (vs. Cache-based, RPC-ARM, and Cache+RPC) or comparable (vs. RPC) latency for a single memory node and 2.2–29% lower latency (vs. RPC) for multi-memory nodes.

Bibliography

[1] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee. MIND: In-Network Memory Management for Disaggregated Data Centers. In *SOSP*, 2021.

[2] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *OSDI*, 2018.

[3] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. In *OSDI*, 2016.

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

[5] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *ASPLOS*, 2014.

[6] L. Liu, W. Cao, S. Sahin, Q. Zhang, J. Bae, and Y. Wu. Memory Disaggregation: Research Problems and Opportunities. In *ICDCS*, 2019.

[7] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *ISCA*, 2009.

[8] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level Implications of Disaggregated Memory. In *HPCA*, 2012.

[9] A. Samih, R. Wang, C. Maciocco, M. Kharbutli, and Y. Solihin. *Collaborative Memories in Clusters: Opportunities and Challenges*. 2014.

[10] Compute Express Link (CXL). https://www.computeexpresslink.org/.

[11] Y. Tang, P. Zhou, W. Zhang, H. Hu, Q. Yang, H. Xiang, T. Liu, J. Shan, R. Huang, C. Zhao, C. Chen, H. Zhang, F. Liu, S. Zhang, X. Ding, and J. Chen. Exploring performance and cost optimization with asic-based cxl memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 818–833, New York, NY, USA, 2024. Association for Computing Machinery.

[12] P. S. Rao and G. Porter. Is memory disaggregation feasible? a case study with spark sql. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, ANCS '16, page 75–80, New York, NY, USA, 2016. Association for Computing Machinery.

[13] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[14] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[15] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 1–16, New York, NY, USA, 2016. Association for Computing Machinery.

[16] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, page 267–280, New York, NY, USA, 2010. Association for Computing Machinery.

[17] M. R. Hines, A. Gordon, M. Silva, D. Da Silva, K. Ryu, and M. Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 130–137, 2011.

[18] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA, December 2008. USENIX Association.

[19] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. *SIGCOMM Comput. Commun. Rev.*, 42(4):431–442, aug 2012.

[20] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[21] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. *SIGCOMM Comput. Commun. Rev.*, 44(4):455–466, aug 2014.

[22] E. Amaro, S. Wang, A. Panda, and M. K. Aguilera. Logical memory pools: Flexible and local disaggregated memory. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 25–32, 2023.

[23] C. Hu, H. Huang, J. Hu, J. Xu, X. Chen, T. Xie, C. Wang, S. Wang, Y. Bao, N. Sun, et al. Memserve: Context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565*, 2024.

[24] Redis. . https://redis.io/.

[25] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS OSR*, 2010.

[26] X. Zhang, U. Khanal, X. Zhao, and S. Ficklin. Making sense of performance in in-memory computing frameworks for scientific data analysis: A case study of the spark system. *J. Parallel Distrib. Comput.*, 120(C):369–382, oct 2018.

[27] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *NSDI*, 2014.

[28] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-Performance, Application-Integrated far memory. In *OSDI*, 2020.

[29] Q. Wang, Y. Lu, and J. Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In *SIGMOD*, 2022.

[30] P. Zuo, J. Sun, L. Yang, S. Zhang, and Y. Hua. One-sided {RDMA-Conscious} extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 15–29, 2021.

[31] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can Far Memory Improve Job Throughput? In *EuroSys*, 2020.

[32] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient Memory Disaggregation with Infiniswap. In *NSDI*, 2017.

[33] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 79–92, 2021.

[34] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A {Memory-Disaggregated} managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280, 2020.

[35] CHASE: Accelerating Distributed Pointer-Traversals on Disaggregated Memory. https://arxiv.org/pdf/2305.02388.pdf, 2023.

[36] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. Stoica. Jiffy: Elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 697–713, New York, NY, USA, 2022. Association for Computing Machinery.

[37] H. Li, D. S. Berger, S. Novakovic, L. R. Hsu, D. Ernst, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2022.

[38] Y. Tang, P. Zhou, W. Zhang, H. Hu, Q. Yang, H. Xiang, T. Liu, J. Shan, R. Huang, C. Zhao, C. Chen, H. Zhang, F. Liu, S. Zhang, X. Ding, and J. Chen. Exploring performance and cost optimization with asic-based cxl memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 818–833, New York, NY, USA, 2024. Association for Computing Machinery.

[39] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. Building an elastic query engine on disaggregated storage. In *USENIX Networked Systems Design and Implementation (USENIX NSDI'20)*.

[40] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 131–141, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 193–206.

[42] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[43] Y. Kim and J. Lin. Serverless data analytics with Flint. In *IEEE International Conference on Cloud Computing (CLOUD '18)*, pages 451–455. IEEE.

[44] Qubole Announces Apache Spark on AWS Lambda. https://www.qubole.com/blog/spark-on-aws-lambda.

[45] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 13–24, New York, NY, USA, 2019. Association for Computing Machinery.

[46] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 363–376.

[47] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451. ACM, 2017.

[48] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley. numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679*, 2018.

[49] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *USENIX Annual Technical Conference (USENIX ATC'19)*.

[50] Amazon. Amazon Athena. https://aws.amazon.com/athena.

[51] Amazon. Amazon Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless.

[52] Azure. Azure SQL Data Warehouse. https://azure.microsoft.com/en-us/services/sql-data-warehouse.

[53] V. Sreekanti, C. W. X. C. Lin, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[54] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica. Caerus: NIMBLE task scheduling for serverless analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 653–669. USENIX Association, April 2021.

[55] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, February 2019. USENIX Association.

[56] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 449–462, Santa Clara, CA, February 2020. USENIX Association.

[57] K. Mahajan, M. Chowdhury, A. Akella, and S. Chawla. Dynamic query Re-Planning using QOOP. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 253–267, Carlsbad, CA, October 2018. USENIX Association.

[58] AWS Lamda. https://aws.amazon.com/lambda/.

[59] Google Cloud Functions. https://cloud.google.com/functions.

[60] Azure Functions. https://azure.microsoft.com/en-us/services/functions.

[61] Hadoop Distributed File System. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[62] C. Gray and D. Cheriton. *Leases: An efficient fault-tolerant mechanism for distributed file cache consistency*, volume 23. ACM, 1989.

[63] M. Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 335–350, 2006.

[64] R. Droms. RFC 2131: Dynamic Host Configuration Protocol. https://www.ietf.org/rfc/rfc2131.txt, 1997.

[65] Amazon ElastiCache. https://aws.amazon.com/elasticache.

[66] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[67] Apache Spark. Unified engine for large-scale data analytics. https://spark.apache.org/.

[68] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[69] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: continuous reliable distributed processing of big data streams. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, pages 439–453.

[70] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *OSDI*, 2010.

[71] MongoDB: Sharded Cluster Balancer. https://docs.mongodb.com/manual/core/sharding-balancer-administration/#sharding-migration-thresholds.

[72] Ceph: Dynamic Bucket Index Resharding. https://docs.ceph.com/docs/mimic/radosgw/dynamicresharding/.

[73] Amazon Simple Notification Service (SNS). https://aws.amazon.com/sns.

[74] R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 7–7, 2004.

[75] Apache Hadoop. https://hadoop.apache.org/.

[76] Amazon EC2. https://aws.amazon.com/ec2/.

[77] Amazon S3. https://aws.amazon.com/s3.

[78] Intel. *Barefoot Networks Unveils Tofino 2, the Next Generation of the World's First Fully P4-Programmable Network Switch ASICs*, 2018. https://bit.ly/3gmZkBG.

[79] EX9200 Programmable Network Switch - Juniper Networks. https://www.juniper.net/us/en/products-services/switching/ex-series/ex9200/.

[80] RoCE vs. iWARP Competitive Analysis. https://www.mellanox.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf, 2017.

[81] MySQL: Adaptive Hash Index. https://dev.mysql.com/doc/refman/8.0/en/innodb-adaptive-hash.html.

[82] SQLServer: Hash Indexes. https://docs.microsoft.com/en-us/sql/database-engine/hash-indexes?view=sql-server-2014.

[83] Teradata: Hash Indexes. https://docs.teradata.com/reader/RtERtp_2wVEQWNxcM3k88w/HmFinSvPP6cTIT6o9F8ZAg.

[84] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling Queries on Compressed Data. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[85] N. Askitis and R. Sinha. HAT-trie: A Cache-conscious Trie-based Data Structure for Strings. In *ACSC*, 2007.

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

[87] A. Braginsky and E. Petrank. A lock-free b+tree. In *SPAA*, 2012.

[88] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: a fast, efficient data structure for string keys. *TOIS*, 2002.

[89] A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *NSDI*, 2016.

[90] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *JACM*, 1968.

[91] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. Surf: Practical range query filtering with fast succinct tries. In *SIGMOD*, 2018.

[92] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.

[93] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*, 2014.

[94] A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*, 2012.

[95] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, 1999.

[96] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into data at facebook. *PVLDB*, 6(11), 2013.

[97] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *OSDI*, 2020.

[98] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *ATC*, 2013.

[99] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *NSDI*, 2013.

[100] X. Shi, S. Pruett, K. Doherty, J. Han, D. Petrov, J. Carrig, J. Hugg, and N. Bronson. Flight-Tracker: Consistency across Read-Optimized online stores at facebook. In *OSDI*, 2020.

[101] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *OSDI*, 2020.

[102] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35, 01 2012.

[103] VoltDB. http://voltdb.com/downloads/datasheets_collateral/technical_overview.pdf.

[104] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, page 371–384, USA, 2013. USENIX Association.

[105] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment*, 8, 11 2014.

[106] MemCached. http://www.memcached.org.

[107] WiredTiger Storage Engine. https://www.mongodb.com/docs/manual/core/wiredtiger/.

[108] M. P. Andersen and D. E. Culler. BTrDB: Optimizing storage system design for timeseries processing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 39–52, Santa Clara, CA, February 2016. USENIX Association.

[109] S.-Y. Tsai, Y. Shan, and Y. Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *ATC*, 2020.

[110] Z. Yu, Y. Zhang, V. Bravermann, M. Chowdhury, and X. Jin. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *SIGCOMM*, 2009.

[111] Standard containers. https://cplusplus.com/reference/stl/.

[112] Boost library. https://www.boost.org/.

[113] Java iterator. https://www.w3schools.com/java/java_iterator.asp.

[114] C++ std::iterator. https://en.cppreference.com/w/cpp/iterator/iterator.

[115] The LLVM Compiler Infrastructure. https://llvm.org/.

[116] A. Rivitti, R. Bifulco, A. Tulumello, M. Bonola, and S. Pontarelli. ehdl: Turning ebpf/xdp programs into hardware designs for the nic. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 208–223, 2023.

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

[118] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[119] LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html#introduction.

[120] K. Koukos, D. Black-Schaffer, V. Spiliopoulos, and S. Kaxiras. Towards more efficient execution: A decoupled access-execute approach. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, page 253–262, New York, NY, USA, 2013. Association for Computing Machinery.

[121] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal. Range translations for fast virtual memory. *IEEE Micro*, 36(3):118–126, 2016.

[122] Xilinx Content Addressable Memory (CAM). https://www.xilinx.com/products/intellectual-property/ef-di-cam.html.

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

[124] AXI4 Protocol Burst size. https://bit.ly/3Bxh35b.

[125] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[126] 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.

[127] Intel(R) RDT Software Package. https://github.com/intel/intel-cmt-cat.

[128] NIVIDIA MELLANOX BLUEFIELD-2. https://network.nvidia.com/files/doc-2020/pb-bluefield-2-smart-nic-eth.pdf.

[129] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[130] WiredTiger storage engine. https://docs.mongodb.com/manual/core/wiredtiger/.

[131] Y. Zhong, H. Li, Y. J. Wu, I. Zarkadas, J. Tao, E. Mesterhazy, M. Makris, J. Yang, A. Tai, R. Stutsman, and A. Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

[132] A Multi-Resolution Plotter that is compatible with BTrDB. https://github.com/BTrDB/mr-plotter.

[133] E. M. Stewart, A. Liao, and C. Roberts. Open μpmu: A real world reference distribution micro-phasor measurement unit data set for research and application development. 2016.

[134] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[135] Xilinx Runtime Library (XRT). https://www.xilinx.com/products/design-tools/vitis/xrt.html.

[136] Running Average Power Limit – RAPL. https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl.

[137] armv8registers. https://developer.arm.com/documentation/100095/0002/system-control/aarch64-register-summary/aarch64-performance-monitors-registers.

[138] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *ASPLOS*, 2022.

[139] CZ120 memory expansion module. https://www.micron.com/products/memory/cxl-memory.

[140] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, page 21–30, New York, NY, USA, 2006. Association for Computing Machinery.

[141] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 105–117, 2015.

[142] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim. Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems. In *2016 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[143] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):640–653, 2018.

[144] F. Schuiki, M. Schaffner, F. K. Gürkaynak, and L. Benini. A scalable near-memory architecture for training deep neural networks on large in-memory datasets. *IEEE Transactions on Computers*, 68(4):484–497, 2018.

[145] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun. Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems*, 67:28–41, 2019.

[146] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann. Livia: Data-centric computing throughout the memory hierarchy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 417–433, 2020.

[147] F. Tu, Y. Wang, Z. Wu, L. Liang, Y. Ding, B. Kim, L. Liu, S. Wei, Y. Xie, and S. Yin. Redcim: Reconfigurable digital computing-in-memory processor with unified fp/int pipeline for cloud ai acceleration. *IEEE Journal of Solid-State Circuits*, 58(1):243–255, 2022.

[148] A. Devic, S. B. Rai, A. Sivasubramaniam, A. Akel, S. Eilert, and J. Eno. To PIM or not for emerging general purpose processing in DDR memory systems. In *ISCA*, pages 231–244, 2022.

[149] Z. Wang, J. Weng, S. Liu, and T. Nowatzki. Near-stream computing: General and transparent near-cache acceleration. In *HPCA*, pages 331–345, 2022.

[150] X. Xie, P. Gu, Y. Ding, D. Niu, H. Zheng, and Y. Xie. Mpu: Memory-centric simt processor via in-dram near-bank computing. *ACM Transactions on Architecture and Code Optimization*, 20(3):1–26, 2023.

[151] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun. A modern primer on processing in memory. In *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*, pages 171–243. Springer, 2022.

[152] G. F. Oliveira, J. Gómez-Luna, S. Ghose, A. Boroumand, and O. Mutlu. Accelerating neural network inference with processing-in-dram: from the edge to the cloud. *IEEE Micro*, 42(6):25–38, 2022.

[153] C. Eckert, A. Subramaniyan, X. Wang, C. Augustine, R. Iyer, and R. Das. Eidetic: An in-memory matrix multiplication accelerator for neural networks. *IEEE Transactions on Computers*, 2022.

[154] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM SIGARCH Computer Architecture News*, 44(3):27–39, 2016.

[155] V. Seshadri and O. Mutlu. Simple operations in memory to reduce data movement. In *Advances in Computers*, volume 106, pages 107–166. Elsevier, 2017.

[156] Y. Kwon, Y. Lee, and M. Rhu. TensorDIMM: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *MICRO*, pages 740–753, 2019.

[157] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng, and O. Mutlu. CoNDA: Efficient cache coherence support for near-Data accelerators. In *ISCA*, pages 629–642, 2019.

[158] B. Y. Cho, Y. Kwon, S. Lym, and M. Erez. Near data acceleration with concurrent host access. In *ISCA*, pages 818–831, 2020.

[159] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang. RecNMP: Accelerating personalized recommendation with near-memory processing. In *ISCA*, pages 790–803, 2020.

[160] Z. Wang, J. Weng, J. Lowe-Power, J. Gaur, and T. Nowatzki. Stream floating: Enabling proactive and decentralized cache optimizations. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 640–653. IEEE, 2021.

[161] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang, X. Hu, and Y. Xie. Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 570–583. IEEE, 2021.

[162] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon, et al. Near-memory processing in action: Accelerating personalized recommendation with axdimm. *IEEE Micro*, 42(1):116–127, 2021.

[163] G. Singh, M. Alser, D. S. Cali, D. Diamantopoulos, J. Gómez-Luna, H. Corporaal, and O. Mutlu. Fpga-based near-memory acceleration of modern data-intensive applications. *IEEE Micro*, 41(4):39–48, 2021.

[164] A. Olgun, J. G. Luna, K. Kanellopoulos, B. Salami, H. Hassan, O. Ergin, and O. Mutlu. Pidram: A holistic end-to-end fpga-based framework for processing-in-dram. *ACM Transactions on Architecture and Code Optimization*, 20(1):1–31, 2022.

[165] G. Dai, Z. Zhu, T. Fu, C. Wei, B. Wang, X. Li, Y. Xie, H. Yang, and Y. Wang. Dimmining: pruning-efficient and parallel graph mining on near-memory-computing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 130–145, 2022.

[166] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, and Y. Xie. ipim: Programmable in-memory image processing accelerator using near-bank architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 804–817. IEEE, 2020.

[167] J. Gómez-Luna, Y. Guo, S. Brocard, J. Legriel, R. Cimadomo, G. F. Oliveira, G. Singh, and O. Mutlu. Evaluating machine learning workloads on memory-centric computing systems. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–49. IEEE, 2023.

[168] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 468–479, New York, NY, USA, 2013. Association for Computing Machinery.

[169] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *International Conference on Computer Design (ICCD)*, 2016.

[170] A. Bhardwaj, C. Kulkarni, and R. Stutsman. Adaptive placement for in-memory storage functions. In *ATC*, 2020.

[171] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman. Splinter: Bare-Metal extensions for Multi-Tenant Low-Latency storage. In *OSDI*, 2018.

[172] J. You, J. Wu, X. Jin, and M. Chowdhury. Ship Compute or Ship Data? Why Not Both? In *NSDI*, pages 633–651, 2021.

[173] S. Novakovic, Y. Shan, A. Kolli, M. Cui, Y. Zhang, H. Eran, B. Pismenny, L. Liss, M. Wei, D. Tsafrir, and M. Aguilera. Storm: A Fast Transactional Dataplane for Remote Data Structures. In *SYSTOR*, page 97–108, 2019.

[174] Q. Zhang, X. Chen, S. Sankhe, Z. Zheng, K. Zhong, S. Angel, A. Chen, V. Liu, and B. T. Loo. Optimizing data-intensive systems in disaggregated data centers with TELEPORT. In *SIGMOD*, pages 1345–1359, 2022.

[175] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. Strom: Smart remote memory. In *EuroSys*, 2020.

[176] J. Shen, P. Zuo, X. Luo, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu. FUSEE: A fully Memory-Disaggregated Key-Value store. In *USENIX FAST*, 2023.

[177] P. Li, Y. Hua, P. Zuo, Z. Chen, and J. Sheng. ROLEX: A scalable RDMA-oriented learned Key-Value store for disaggregated memory systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 99–114, Santa Clara, CA, February 2023. USENIX Association.

[178] H. An, F. Wang, D. Feng, X. Zou, Z. Liu, and J. Zhang. Marlin: A concurrent and write-optimized b+-tree index on disaggregated memory. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 695–704, New York, NY, USA, 2023. Association for Computing Machinery.

[179] X. Min, K. Lu, P. Liu, J. Wan, C. Xie, D. Wang, T. Yao, and H. Wu. Sephash: A write-optimized hash index on disaggregated memory via separate segment structure. *Proc. VLDB Endow.*, 17(5):1091–1104, 2024.

[180] J. Shen, P. Zuo, X. Luo, Y. Su, J. Gu, H. Feng, Y. Zhou, and M. R. Lyu. Ditto: An elastic and adaptive memory-disaggregated caching system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 675–691, New York, NY, USA, 2023. Association for Computing Machinery.

[181] D. Korolija, T. Roscoe, and G. Alonso. Do OS abstractions make sense on FPGAs? In *OSDI*, 2020.

[182] E. Amaro, Z. Luo, A. Ousterhout, A. Krishnamurthy, A. Panda, S. Ratnasamy, and S. Shenker. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 38–44, 2020.

[183] W. Reda, M. Canini, D. Kostić, and S. Peter. RDMA is turing complete, we just did not know it yet! In *NSDI*, 2022.

[184] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, I. Jeong, R. Wang, and N. S. Kim. Demystifying cxl memory with genuine cxl-ready systems and devices, 2023.

[185] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, I. Agarwal, M. Hill, M. Fontoura, et al. First-generation memory disaggregation for cloud platforms. *arXiv preprint arXiv:2203.00241*, 2022.

[186] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[187] A. Cho, A. Saxena, M. Qureshi, and A. Daglis. A case for cxl-centric server processors, 2023.

[188] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, I. Jeong, R. Wang, and N. S. Kim. Demystifying cxl memory with genuine cxl-ready systems and devices, 2023.

[189] Intel Corporation. Intel Agilex® 7 FPGA and SoC FPGA I-Series. https://www.intel.com/content/www/us/en/products/details/fpga/agilex/7/i-series.html.

[190] V. Viswanathan, K. Kumar, and T. Willhalm. "Intel® Memory Latency Checker v3.10". https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[191] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[192] D. Blanchfield. The cloud native convergence: A new era of data-intensive applications. https://elnion.com/2023/06/05/the-cloud-native-convergence-a-new-era-of-data-intensive-applications/.

[193] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 971–985, New York, NY, USA, 2019. Association for Computing Machinery.

[194] S.-P. Yang, M. Kim, S. Nam, J. Park, J. yong Choi, E. H. Nam, E. Lee, S. Lee, and B. S. Kim. Overcoming the memory wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 601–617, Boston, MA, July 2023. USENIX Association.

[195] Leo Memory Connectivity Platform for CXL 1.1 and 2.0. https://www.asteralabs.com/wp-content/uploads/2022/08/Astera_Labs_Leo_Aurora_Product_FINAL.pdf.

[196] D. S. Berger, D. Ernst, H. Li, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, L. Hsu, I. Agarwal, M. D. Hill, and R. Bianchini. Design tradeoffs in cxl-based memory pools for public cloud platforms. *IEEE Micro*, 43(2):30–38, 2023.

[197] D. Gouk, S. Lee, M. Kwon, and M. Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[198] K. Kim, H. Kim, J. So, W. Lee, J. Im, S. Park, J. Cho, and H. Song. Smt: Software-defined memory tiering for heterogeneous computing systems with cxl memory expander. *IEEE Micro*, 43(2):20–29, 2023.

[199] D. S. Berger, D. Ernst, H. Li, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, L. Hsu, I. Agarwal, M. D. Hill, and R. Bianchini. Design tradeoffs in cxl-based memory pools for public cloud platforms. *IEEE Micro*, 43(2):30–38, 2023.

[200] What Are PCIe 4.0 and 5.0? https://www.intel.com/content/www/us/en/gaming/resources/what-is-pcie-4-and-why-does-it-matter.html.

[201] D. D. Sharma, R. Blankenship, and D. S. Berger. An introduction to the compute express link (cxl) interconnect, 2023.

[202] Intel Corporation. Intel launches 4^{th} gen xeon scalable processors, max series cpus. https://www.intel.com/content/www/us/en/newsroom/news/.

[203] AMD Unveils Zen 4 CPU Roadmap: 96-Core 5nm Genoa in 2022, 128-Core Bergamo in 2023. https://wccftech.com/intel-clearwater-forest-e-core-xeon-cpus-up-to-288-cores-higher-ipc-more-cache/.

[204] Montage Technology. Cxl memory expander controller (mxc). https://www.montage-tech.com/MXC,accessedin2023.

[205] M. Ahn, A. Chang, D. Lee, J. Gim, J. Kim, J. Jung, O. Rebholz, V. Pham, K. Malladi, and Y. S. Ki. Enabling cxl memory expansion for in-memory database management systems. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*, DaMoN '22, New York, NY, USA, 2022. Association for Computing Machinery.

[206] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, page 21–30, New York, NY, USA, 2006. Association for Computing Machinery.

[207] J. Weiner. [PATCH] mm: mempolicy: N:M interleave policy for tiered memory nodes. https://lore.kernel.org/linux-mm/YqD0%2FtzFwXvJ1gK6@cmpxchg.org/T/.

[208] NUMA balancing: optimize memory placement for memory tiering system. https://lore.kernel.org/linux-mm/20220221084529.1052339-1-ying.huang@intel.com/.

[209] Transparent Page Placement for Tiered-Memory. https://lore.kernel.org/all/cover.1637778851.git.hasanalmaruf@fb.com/.

[210] David L Mulnix. Intel® Xeon® Processor Scalable Family Technical Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html.

[211] A. Cho and et al. A Case for CXL-Centric Server Processors. https://arxiv.org/abs/2305.05033.

[212] J. Yi, B. Dong, M. Dong, R. Tong, and H. Chen. MT$\hat{2}$: Memory bandwidth regulation on hybrid NVM/DRAM platforms. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 199–216, Santa Clara, CA, February 2022. USENIX Association.

[213] Intel Corporation. Intel® Performance Counter Monitor (Intel® PCM). https://github.com/intel/pcm.

[214] Intel Corporation. Intel Unveils Future-Generation Xeon with Robust Performance and Efficiency Architectures. https://www.intel.com/content/www/us/en/newsroom/news/intel-unveils-future-generation-xeon.html.

[215] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[216] Tecton.ai. Managing your Redis Cluster. https://docs.tecton.ai/docs/0.5/setting-up-tecton/setting-up-other-components/managing-your-redis-cluster.

[217] Google Cloud. Memory management best practices. https://cloud.google.com/memorystore/docs/redis/memory-management-best-practices.

[218] Redis enterprise. https://redis.io/docs/about/redis-enterprise/, 2023.

[219] Auto Tiering Extend Redis Enterprise databases beyond DRAM limits. https://redis.com/redis-enterprise/technology/auto-tiering/#:~:text=Redis%20Enterprise's%20auto%20tiering%20lets,compared%20to%20only%20DRAM%20deployments.

[220] C. Zou, H. Zhang, A. A. Chien, and Y. Seok Ki. Psacs: Highly-parallel shuffle accelerator on computational storage. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 480–487, 2021.

[221] TPC-H is a Decision Support Benchmark. https://www.tpc.org/tpch/.

[222] Ice Lake SP: Overview and technical documentation. (n.d.). Intel. https://www.intel.com/content/www/us/en/products/platforms/details/ice-lake-sp.html.

[223] 4th Gen Intel Xeon Processor Scalable Family, sapphire rapids. (n.d.). Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html#gs.3m5uv2.

[224] McDowell, S. (2023, December 18). Intel launches 5th generation "Emerald Rapids" Xeon processors. Forbes. https://www.forbes.com/sites/stevemcdowell/2023/12/17/intel-launches-5th-generation-emerald-rapids-xeon-processors/.

[225] Kennedy, Patrick. "Intel Shows Granite Rapids and Sierra Forest Motherboards at OCP Summit 2023." ServeTheHome, 26 Oct. 2023,. www.servethehome.com/intel-shows-granite-rapids-and-sierra-forest-motherboards-at-ocp-summit-2023-qct-wistron.

[226] Mujtaba, H. (2023, December 1). Intel Clearwater Forest E-Core Only Xeon CPUs to offer up to 288 cores. https://wccftech.com/intel-clearwater-forest-e-core-xeon-cpus-up-to-288-cores-higher-ipc-more-cache/.

[227] S. Yi, D. Kondo, and A. Andrzejak. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 236–243, 2010.

[228] Amazon EC2 M7a Instances. https://aws.amazon.com/ec2/instance-types/m7a/, 2023.

[229] Amazon EC2 M7i Instances. https://aws.amazon.com/ec2/instance-types/m7i/, 2023.

[230] Intel Shows Granite Rapids and Sierra Forest Motherboards at OCP Summit 2023. https://www.servethehome.com/intel-shows-granite-rapids-and-sierra-forest-motherboards-at-ocp-summit-2023-qct-wistron/.

[231] Elastic Compute Service, Volcano Engine, Bytedance. https://www.volcengine.com/product/ecs.

[232] G. W. D. Patel. GPT-4 Architecture, Infrastructure, Training Dataset, Costs, Vision, MoE. https://www.semianalysis.com/p/gpt-4-architecture-infrastructure, 2023.

[233] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention, 2023.

[234] Lightllm A Light and Fast Inference Service for LLM. https://github.com/ModelTC/lightllm.

[235] Julien Simon.Smaller is Better: Q8-Chat LLM is an Efficient Generative AI Experience on Intel® Xeon® Processors. https://www.intel.com/content/www/us/en/developer/articles/case-study/q8-chat-efficient-generative-ai-experience-xeon.html.

[236] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean. Efficiently scaling transformer inference, 2022.

[237] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[238] Alpaca: A Strong, Replicable Instruction-Following Model. https://crfm.stanford.edu/2023/03/13/alpaca.html.

[239] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[240] L. Floridi and M. Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020.

[241] K. Ethayarajh. How contextual are contextualized word representations? comparing the geometry of bert, elmo, and gpt-2 embeddings. *arXiv preprint arXiv:1909.00512*, 2019.

[242] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[243] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, Santa Clara, CA, July 2024. USENIX Association.

[244] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[245] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[246] C. Hu, H. Huang, J. Hu, J. Xu, X. Chen, T. Xie, C. Wang, S. Wang, Y. Bao, N. Sun, and Y. Shan. Memserve: Context caching for disaggregated llm serving with elastic memory pool, 2024.

[247] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, H. Jin, T. Chen, and Z. Jia. Towards efficient generative large language model serving: A survey from algorithms to systems. *arXiv preprint arXiv:2312.15234*, 2023.

[248] J. Yao, H. Li, Y. Liu, S. Ray, Y. Cheng, Q. Zhang, K. Du, S. Lu, and J. Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion, 2024.

[249] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang. Flexgen: High-throughput generative inference of large language models with a

single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[250] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[251] Y. Liu, H. Li, K. Du, J. Yao, Y. Cheng, Y. Huang, S. Lu, M. Maire, H. Hoffmann, A. Holtzman, et al. Cachegen: Fast context loading for language model applications. *arXiv preprint arXiv:2310.07240*, 2023.

[252] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 105–121, New York, NY, USA, 2023. Association for Computing Machinery.

[253] Y. Tang, P. Zhou, W. Zhang, H. Hu, Q. Yang, H. Xiang, T. Liu, J. Shan, R. Huang, C. Zhao, C. Chen, H. Zhang, F. Liu, S. Zhang, X. Ding, and J. Chen. Exploring performance and cost optimization with asic-based cxl memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 818–833, New York, NY, USA, 2024. Association for Computing Machinery.

[254] P. Labs. GPT-fast Simple and efficient pytorch-native transformer text generation. https://github.com/pytorch-labs/gpt-fast.git, 2024.

[255] Y. Leviathan, M. Kalman, and Y. Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[256] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[257] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[258] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[259] I. Corporation. Intel® Xeon® Platinum Processor. https://www.intel.com/content/www/us/en/products/details/processors/xeon/scalable/platinum.html, 2024.

[260] M. Arif, A. Maurya, and M. M. Rafique. Accelerating performance of gpu-based workloads using cxl. In *Proceedings of the 13th Workshop on AI and Scientific Computing at Scale Using Flexible Computing*, FlexScience '23, page 27–31, New York, NY, USA, 2023. Association for Computing Machinery.

[261] S. Sano, Y. Bando, K. Hiwada, H. Kajihara, T. Suzuki, Y. Nakanishi, D. Taki, A. Kaneko, and T. Shiozawa. Gpu graph processing on cxl-based microsecond-latency external memory. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 962–972, New York, NY, USA, 2023. Association for Computing Machinery.

[262] D. Gouk, S. Kang, H. Bae, E. Ryu, S. Lee, D. Kim, J. Jang, and M. Jung. Breaking barriers: Expanding gpu memory with sub-two digit nanosecond latency cxl controller. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '24, page 108–115, New York, NY, USA, 2024. Association for Computing Machinery.

[263] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[264] NVIDIA. CUDA Runtime API - Memory Management Functions. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html, 2024.

[265] anon8231489123. ShareGPT Vicuna unfiltered dataset. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered, 2024.

[266] Meta. Llama 2 70B: An MLPerf Inference Benchmark for Large Language Models. https://mlcommons.org/2024/03/mlperf-llama2-70b, 2024.