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 [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. 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 [**?**, **?**, **?**], CXL [**?**, **?**], 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 [**?**, **?**, **?**, **?**, **?**, **?**] plays a crucial and foundational role. In traditional monolithic server configurations, memory often becomes a bottleneck, limiting the scalability and adaptability of applications. This issue has been frequently observed and reported in production data centers [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. By decoupling memory resources from compute and storage elements and presenting them as pooled, disaggregated resources [**?**, **?**], data centers can achieve increased efficiency, scalability, and adaptability. Memory-intensive applications [**?**, **?**, **?**] 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. This ultimately leads to improved performance and better resource utilization.

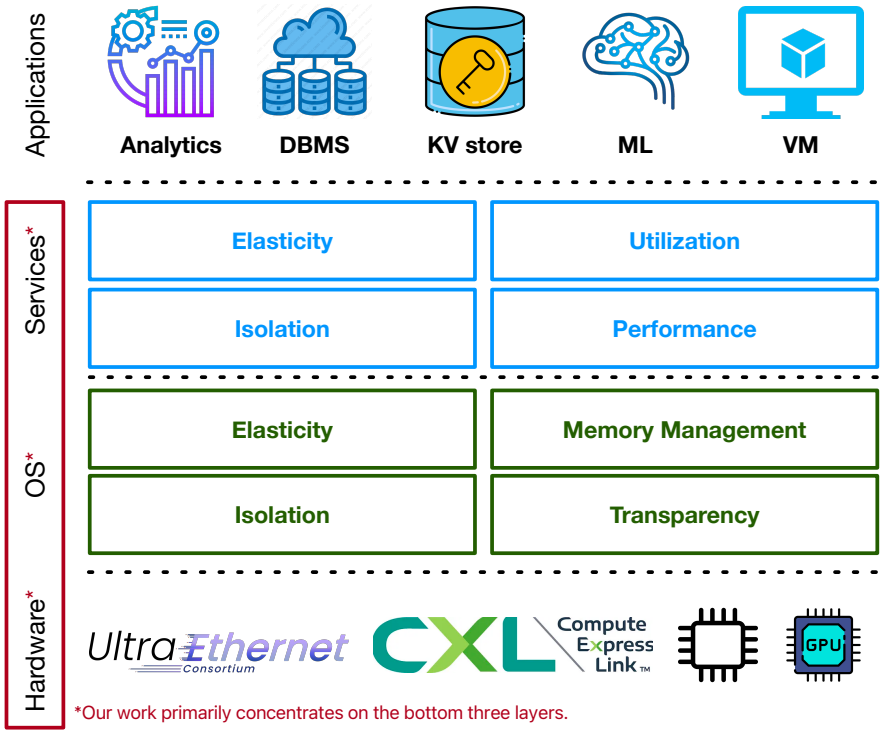


Fig. 1.1: **Cloud Stack of Disaggregated Architecture.**

1.1 Limitations of Existing Approaches

While resource disaggregation offers numerous advantages, transitioning existing applications to a disaggregated architecture is far from straightforward. Recent research has explored various approaches to address this challenge. Some efforts focus on adapting applications to optimize their use of disaggregated memory [**?**, **?**, **?**, **?**], while others aim to transparently port applications, shifting the responsibility of mitigating performance penalties—arising from the mismatch between disaggregated architectures and traditional software interfaces—to the service or operating system layer [**?**, **?**, **?**, **?**, **?**, **?**].

The core issue lies in the fundamental mismatch between the existing cloud stack, designed for monolithic architectures, and the requirements of disaggregated architectures (Figure **??**). 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 face unique challenges compared to traditional monolithic systems. The primary difference is resource distribution: compute, memory, and storage are spread across multiple nodes instead of centralized in one server. This

requires complex communication and data management strategies to handle increased latency and resource management needs. In contrast, monolithic architectures offer integrated resources, simplifying application interaction. Adapting to disaggregated systems involves significantly redesigning applications for effective resource utilization and management.

OS support. Unlike monolithic servers where the 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 [**?**] in the architecture or disaggregating its functions across different resource nodes [**?**].

Performance overheads of disaggregation. Transitioning existing applications to a disaggregated architecture transparently introduces a spectrum of performance challenges. These include, but are not limited to, managing memory partitioning [**?**] and addressing applications with irregular memory access patterns [**?**]. 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. Using networks as interconnects for resource disaggregation has been a subject of exploration in academia and industry. However, networks have inherent challenges, such as performance slowdowns compared to intra-server resource access and a lack of inherent coherency. Advanced hardware technologies like Compute Express Link (CXL) [**?**, **?**, **?**] offer promising enhancements with faster access times and hardware-supported cache coherence. Yet, the current state of hardware prototypes and software support for these technologies remains limited.

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 Memory management as a Service

With least modification to lower layers such as OS/Hardware, we explore the design requirement and challenges in provoding memory management as a service. We proposed an end-to-end system design called Jiffy, which enables multiple application/tasks multiplex memory in a elastic manner.

Jiffy also provides multiple popular data structure interface and can be easily applied to existing cloud applications.

1.2.2 In-network memory management OS-design

As we decouple compute and memory resources in disaggregated architecture. There is no single host as if in monolithic architecture in order to implement the key unit of resource management - the operating system. We proposal a new generation operating system design by placing OS functionality inside the interconnects. We start by a system called MIND, addressing the basic problems in memory management, such as memory address translation, memory protection, and cache coherence between multiple hosts. Such resource decoupling and in-network memory management serves well for cache-friendly workload, but performs poor for cache-friendly workload due to the back-and-forth communication over the slower interconnects. We then develop optimizations for dealing with cache-unfriendly workloads. We design and implement a near memory accelerator from scratch, named PULSE. PULSE analyzes popular pointer traversal applications and identify a common but simple interface that can be easily integrated into existing cloud applications.

1.2.3 Memory management adaptation for new-generation interconnects

In prior work [**?**, **?**], ethernet is considered as the most popular interconnect for disaggregated data centers. However, as new memory interconnects are emerging, such as Compute Express Link(CXL), new adaptation of memory management needs to be made regarding the new interconnect interface. Within the context of disaggregated architecture, new problems arises such as how can the applications leverage multiple tiers of memory. Therefore, we start with a perfomance analysis on CXL 1.1 single host extended memory, and then we propose a new system design that integrates disaggregated CXL memory pool with today's emerging popular application - LLM inference.

1.3 Outline and Previously Published Work

This dissertation is organized as follows. Chapter **??** introduces Jiffy, a distributed memory management system that decouples memory capacity and lifetime from compute in the serverless paradigm. Chapter **??** 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 **??** presents our exploration in latest Compute Express Link(CXL) hardware. We conclude with our contributions and possible future work directions in Chapter **??**.

Chapter **??** revises material from [**?**]. Chapter **??** revises material from [**?**] and [**?**]. Finally, Chapter **??** revises material from [**?**].

Chapter 2

Memory Management as a Service

The service layer, positioned above the OS layer, plays a pivotal role in facilitating efficient and seamless memory sharing across multiple computing and memory nodes within a disaggregated architecture. As application software, it provides greater flexibility than the operating system, allowing for a variety of services to be offered to applications. These adaptable services enable applications to choose options best suited to their specific needs. However, this requires that the storage and compute are easily decoupled, otherwise the application developers will need to spend enormous effort to modify the application for it to use memory management service.

Serverless architecture offer on-demand elasticity of compute and storage and decouples them logically. Recent work on serverless analytics has demonstrated the benefit of using serverless architecture for resource- and cost-efficient data analytics. The key idea of serverless analytics is to use a remote low-latency, high-throughput shared far-memory system for (1) inter-task communication and (2) for multi-stage jobs, storing intermediate data beyond the lifetime of the task that produced the data. This makes it a perfect target for disaggregate memory since compute and memory are decoupled logically when the serverless task is assigned.

Designing a memory management service is a non-trivial tasks. Our discussion begins with an outline of the essential requirements for such memory management services, focusing on the unique challenges introduced by disaggregation. We then highlight our current efforts to tackle these challenges and explore potential directions for future research in this rapidly evolving domain.

Elasticity. Memory usage in modern computing environments can be highly variable, with ap-

plications experiencing fluctuating memory demands [**?**]. Elasticity allows the memory service to dynamically allocate and deallocate memory resources based on current requirements, optimizing resource utilization. In typical applications with dynamic memory requirements, such as data analytics, applications are organized into jobs that contain multiple tasks. Each task can be assigned to run on an arbitrary compute node. Each task communicates with the other using memory as intermediate storage. Previous solutions [**?**] tend to allocate resources in a job granularity. Jobs specify their memory demands before the job is submitted and the system reserves the amount of memory for the entire job lifetime. The tradeoff between performance and resource utilization for such job-level resource allocation is indeed well studied in prior work [**?**]. On the one hand, if jobs specify an average demand of memory, the job will degrade as running out of memory will lead to swapping data out to slower storage medium (e.g. S3 storage), while on the other hand allocating at peak granularity will result in resource wastage.

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 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 Elastic memory management for data analytics

Data analytics applications, which utilize disaggregated memory for inter-task communication and intermediate data storage, are becoming increasingly common. As discussed in [**?**, **?**, **?**, **?**], 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 [**?**, **?**]. 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 [**?**], indicate a significant fluctuation in the ratio of peak to average demands, sometimes varying by two orders of magnitude within minutes.

In response to the challenges of dynamically allocating memory resources in data analytics applications, we have developed Jiffy [**?**], an elastic memory service tailored for disaggregated architectures. As shown in Figure **??**, Jiffy allocates memory in small, fixed-size blocks, enabling the dynamic adjustment of memory allocation for individual jobs without prior knowledge of intermediate data sizes. Jiffy employs a hierarchical address space that reflects the structure of the analytics job, facilitating efficient management of the relationship between memory blocks and tasks while

ensuring task-level isolation.

2.2 Introduction

Serverless architectures offer flexible compute and storage options, charging users for precise resource usage. Initially used for web microservices, IoT, and ETL tasks, recent advancements show their efficacy in data analytics. Serverless analytics leverage remote, high-throughput memory systems for inter-task communication and storing intermediate data. However, existing far-memory systems face limitations, allocating resources at the job level, leading to performance issues and underutilization.

To address this, we introduce Jiffy, an elastic far-memory system for stateful serverless analytics. Unlike conventional systems, Jiffy allocates memory in small, fixed-size blocks, enabling dynamic scaling and efficient resource utilization. This approach resolves challenges unique to serverless analytics, including task mapping, task isolation, and data lifetime management.

Our implementation of Jiffy features an intuitive API for seamless data manipulation. We demonstrate its versatility by implementing popular distributed frameworks like MapReduce, Dryad, StreamScope, and Piccolo. Evaluation against state-of-the-art systems indicates Jiffy's superior resource utilization and application performance, achieving up to 3x better efficiency and 1.6–2.5x performance improvements.

2.3 Motivation

The leading system for stateful serverless analytics is Pocket, a distributed system designed for high-throughput, low-latency storage of intermediate data. Pocket effectively tackles several key challenges in stateful serverless analytics, including:

Centralized management. Pocket's architecture features separate control, metadata, and data planes. While data storage is distributed across multiple servers, management functions are centralized, simplifying resource allocation and storage organization. A single metadata server can handle significant request loads, supporting thousands of serverless tasks.

Multi-tiered data storage. Pocket's data plane stores job data across multiple servers and serves them via a key-value API. It supports storage across different tiers like DRAM, Flash, or HDD, enabling flexibility based on performance and cost constraints.

Dynamic resource management. Pocket can scale memory capacity by adding or removing memory servers based on demand. The controller allocates resources for jobs and informs the metadata plane for proper data placement.

Analytics execution with Pocket. Jobs interact with Pocket by registering with the control plane, specifying memory resources needed. The controller allocates resources and informs the metadata plane. Serverless tasks can access data directly from memory servers. Once a job finishes, it deregisters to release resources.

In our analysis, we focus on challenges in Pocket's resource allocation. Pocket allocates memory at the job level, which poses challenges in accurately predicting intermediate data sizes and leads to performance degradation or resource underutilization. This issue persists due to the dynamic nature of intermediate data sizes across different stages of execution.

2.4 Jiffy Design

2.4.1 Overview

Jiffy facilitates precise sharing of far-memory capacity among concurrent serverless analytics tasks for intermediate data storage. Drawing inspiration from virtual memory, Jiffy divides memory capacity into fixed-sized blocks, akin to virtual memory pages, and performs allocations at this granular level. This approach yields two key benefits: firstly, Jiffy can swiftly adapt to instantaneous job demands, adjusting capacity at the block level within seconds. Secondly, Jiffy doesn't necessitate prior knowledge of intermediate data sizes from jobs; instead, it dynamically manages resources as tasks write or delete data.

It's worth noting that multiplexing available memory capacity differs from merely scaling the memory pool's overall capacity. While prior systems like Pocket focus on the latter, adding or removing memory servers based on job arrivals or completions, Jiffy prioritizes efficient sharing of available capacity among concurrent jobs. This approach minimizes underutilization of existing capacity, a common issue in job-level resource allocation systems. Even during high memory capacity utilization, Jiffy can augment capacity by adding memory servers akin to Pocket. Notably, by efficiently multiplexing capacity across concurrent jobs, Jiffy reduces the need for frequent additions or removals of memory servers.

In addressing the challenges posed by serverless analytics, Jiffy implements hierarchical addressing, data lifetime management, and flexible data repartitioning. These mechanisms are discussed in detail in subsequent sections, with illustrative examples provided in Fig. 3, depicting a typical analytics job's execution plan organized as a directed acyclic graph (DAG) with computation tasks represented as serverless functions exchanging intermediate data via Jiffy.

2.4.2 Hierarchical Addressing

Analytics jobs typically follow a multi-stage or directed acyclic graph structure. In serverless analytics, where compute elasticity is integral, each job may entail tens to thousands of individual tasks. Consequently, achieving fine-grained resource allocation necessitates an efficient mechanism for maintaining an updated mapping between tasks and allocated memory blocks. Additionally, the rapidly changing number of tasks accessing shared memory underscores the importance of isolation at the task level to prevent performance degradation across jobs. In this context, Jiffy's hierarchical addressing system plays a crucial role.

Instead of relying on a network structure, Jiffy employs a hierarchical addressing mechanism tailored to the execution structure of analytics jobs. It organizes intermediate data within a virtual address hierarchy, reflecting the dependencies between tasks in the job's DAG. For instance, internal nodes represent tasks, while leaf nodes denote memory blocks storing intermediate data. The addressing scheme enables precise resource allocation at the task level, independent of other tasks, akin to virtual memory's process-level isolation.

This hierarchical addressing facilitates efficient management of resource allocations, ensuring that overflow into persistent storage doesn't impact the performance of other tasks. Each memory block, once allocated, remains dedicated to its task until explicitly released, guaranteeing isolation at the task level regardless of concurrency. This approach aligns with virtual memory principles, where each process enjoys its own address space, ensuring isolation at the process level.

Jiffy's design considers two key aspects. Firstly, resource allocation is decoupled from policy enforcement, allowing seamless integration of fairness algorithms atop Jiffy's allocation mechanism. Secondly, address translation, handled centrally, enables addressing for arbitrary DAGs without imposing limitations on execution structure complexity. While Jiffy's hierarchical addressing introduces complexity at the controller, its scalability is validated in our evaluation, accommodating

realistic deployment demands.

Regarding block sizing, Jiffy's approach, akin to traditional virtual memory's page sizing, balances metadata overhead and memory utilization. Larger block sizes reduce per-block metadata, but may lead to data fragmentation, while smaller sizes optimize memory utilization at the expense of increased metadata overhead. Jiffy mitigates fragmentation via data repartitioning and allows block size configuration during initialization for compatibility with analytics frameworks.

Isolation granularity in Jiffy is task-level by default, but can be adjusted finer or coarser by adapting the hierarchy. For most analytics frameworks, task-level isolation suffices, but custom hierarchies can be created using Jiffy's API to tailor isolation to specific needs.

2.4.3 Data Lifetime Management

Existing far-memory systems for serverless analytics typically manage data lifetimes at the granularity of entire jobs, reclaiming storage only when a job explicitly deregisters. However, in serverless analytics, the intermediate data of a task is dissociated from its execution, residing in the far-memory system. This decoupling extends to fault domains: traditional mechanisms, such as reference counting, can result in dangling intermediate data if a task fails. To address this inefficiency, effective task-level data lifetime management mechanisms are required.

Jiffy tackles this challenge by integrating lease management mechanisms with hierarchical addressing. Each address-prefix in a job's hierarchical addressing is associated with a lease, and data remains in memory only as long as the lease is renewed. Consequently, jobs periodically renew leases for the address-prefixes of running tasks. Jiffy tracks lease renewal times for each node in the address hierarchy, updating them accordingly. Upon lease expiry, Jiffy reclaims allocated memory after flushing data to persistent storage, ensuring data integrity even in the event of network delays.

A novel aspect of Jiffy's lease management is its utilization of DAG-based hierarchical addressing to determine dependencies between leases. When a task renews its lease, Jiffy extends the renewal to the prefixes of tasks it depends on (parent nodes) and the prefixes of tasks dependent on it (descendant nodes), minimizing the number of renewal messages sent. This approach ensures that not only is a task's own data retained in memory while it's active, but also the data of tasks it depends on and tasks dependent on it. This mechanism strikes a balance between age-based eviction and explicit resource management, granting jobs control over resource lifetimes while tying

resource fate to job status.

In an example scenario, task T7 periodically renews leases for its prefix during execution, ensuring the retention of intermediate data for blocks under it in memory. Lease renewals for T7's prefix also extend to its parent and descendant tasks, ensuring continuity of data access. However, leases for inactive tasks are not automatically renewed, preventing unnecessary resource retention.

Lease duration in Jiffy involves a tradeoff between control plane bandwidth and system utilization. Longer lease durations reduce network traffic but may lead to underutilization of resources until leases expire. Jiffy's sensitivity to lease durations is evaluated in the subsequent section.

2.4.4 Flexible Data Repartitioning

Decoupling compute tasks from their intermediate data in serverless analytics poses a challenge in achieving memory elasticity efficiently at fine granularities. When memory is allocated or deallocated to a task, repartitioning the intermediate data across the remaining memory blocks becomes necessary. However, due to the decoupling and the high concurrency of tasks, it's impractical to expect the application to handle this repartitioning. For instance, in many existing serverless analytics systems, key-value stores are used to store intermediate data. If a compute task were to handle repartitioning upon memory scaling, it would need to fetch key-value pairs from the store over the network, compute new data partitions, and then write back the data, incurring significant network latency and bandwidth overheads.

As discussed in §5, Jiffy already incorporates standard data structures utilized in data analytics frameworks, ranging from files to key-value pairs to queues. Analytics jobs leveraging these data structures can delegate repartitioning of intermediate data upon resource allocation/deallocation to Jiffy. Each block allocated to a Jiffy data structure monitors the fraction of memory capacity currently utilized for data storage. When usage surpasses a high threshold, Jiffy allocates a new block to the corresponding address-prefix. Subsequently, the overloaded block initiates data structure-specific repartitioning to migrate some data to the new block. Conversely, when block usage falls below a low threshold, Jiffy identifies another block with low usage within the address-prefix for potential data merging. The block then undergoes the necessary repartitioning before deallocation by Jiffy.

By tasking the target block with repartitioning instead of the compute task, Jiffy circumvents

network and computational overheads for the task itself. Furthermore, data repartitioning in Jiffy occurs asynchronously, enabling data access operations across data structure blocks to proceed even during repartitioning. This ensures minimal impact on application performance due to repartitioning.

The data structures integrated into Jiffy enable the implementation of serverless versions of various powerful distributed programming frameworks, including MapReduce, Dryad, StreamScope, and Piccolo. Notably, the simplicity of repartitioning mechanisms required by analytics framework data structures allows serverless applications utilizing these programming models to seamlessly run on Jiffy and leverage its adaptable data repartitioning without any modifications.

Regarding thresholds for elastic scaling, the high and low thresholds in Jiffy present a tradeoff between data plane network bandwidth and task performance on one side and system utilization on the other. Optimizing these thresholds balances the frequency of elastic scaling triggers and system utilization efficiency. We evaluate Jiffy's sensitivity to threshold selections in §6.6.

2.5 Implementation

We implement Jiffy based on prior Serverless memory management system - Pocket. We reused the scalable and fault-tolerant metadata plane, system-wide capacity scaling, analytics execution model, etc. However, Jiffy implements hierarchical addressing, lease management and efficient data repartitioning to resolve unique challenges introduced by serverless enviroment.

2.5.1 Jiffy Interface

We describe Jiffy interface in terms of its user-facing API and internal API.

User-facing API. User-facing API. Jiffy's user-facing interface (Table 1) is divided along its two core abstractions: hierarchical addresses and data structures. Jobs add a new address-prefix to their address hierarchy using createAddrPrefix, specifying the parent address-prefix, along with optional arguments such as initial capacity. Jiffy also provides a createHierarchy interface to directly generate the complete address hierarchy from the application's execution plan (i.e., DAG), and flush/load interfaces to persist/load address-prefix data from external storage (e.g., S3). Jiffy provides three built-in data structures that can be associated with an address-prefix (via initDataStructure), and a way to define new data structures using its internal API.

Similar to existing systems, data structures also expose a notification interface, so that tasks that consume intermediate data can be notified on data availability. For instance, a task can subscribe to write operations on its parent task's data structure, and obtain a listener handle. Jiffy asynchronously notifies the listener upon a write to the data structure, which the task can get via listener.get().

Internal API. The data layout within blocks in Jiffy is unique to the data structure that owns it. As such, Jiffy blocks expose a set of data structure operators (Fig. 6) that uniquely define how data structure requests are routed across their blocks and how data is accessed or modified. These operators are used internally within Jiffy for its built-in data structures (§5) and are not exposed to jobs directly.

The getBlock operator determines which block an operation request is routed to based on the operation type and operation-specific arguments (e.g., based on key hashes for a KV-store) and returns a handle to the corresponding block. Each Jiffy block exposes writeOp, readOp, and deleteOp operators to facilitate data structure-specific access logic (e.g., get, put, and delete for KV-store). Jiffy executes individual operators atomically using sequence numbers, but does not support atomic transactions that span multiple operators.

2.6 Implementation

Jiffy's high-level design components are similar to Pocket's, 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 are still applicable to the unified control plane.

2.6.1 Jiffy Controller

The Jiffy controller (Fig. 7) 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 a 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 job IDs (which uniquely identify jobs) and their address hierarchies is stored in a hash table at the controller.

Block allocator. When a job creates an address prefix in Jiffy, the block allocator at the control plane assigns it the number of blocks corresponding to the requested initial capacity from its pool of free blocks. While assigning the blocks, the controller updates its state: the free block list, access permissions, and block-map for that address prefix. Assignment of blocks across address prefixes is akin to virtual memory in traditional operating systems: Jiffy multiplexes its physical memory pools at the data plane across different prefixes at block granularity, while individual tasks operate under the illusion that their prefixes have infinite memory resources.

Metadata manager. The metadata manager tracks the partitioning information specific to different data structures (§5) and assists clients in maintaining a consistent view of how the data is organized across the blocks allocated to each data structure. We defer the discussion of data structure-specific metadata stored at the control plane to §5, but note that this metadata is updated whenever blocks allocated to an address prefix are scaled. A client detects that a scaling has occurred when it queries the data plane and updates its view of the partitioning metadata by querying the control plane.

Lease manager. The lease manager implements lifetime management in Jiffy. It comprises a lease renewal service that 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.

Controller scaling and fault tolerance. In order to scale the control plane, Jiffy can employ multiple controller servers, each managing control operations for a non-overlapping subset of address hierarchies (across jobs) and blocks (across memory servers at the data plane). Jiffy employs hash partitioning to distribute both address prefixes and memory blocks (via their block IDs) across controller servers. Moreover, Jiffy employs the same approach to scale its control plane to multiple cores on a multi-core server. Jiffy adopts primary-backup based mechanisms from prior work [8, 69] at each controller server for fault-tolerance.

2.6.2 Jiffy 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 memory servers across fixed-sized blocks. Each memory server maintains, for the blocks managed by it, a mapping from unique block IDs to pointers to raw memory allocated to the blocks, along with two additional metadata: data structure-specific operator implementations as described in §4.1, and a subscription map that maps data structure operations to client handles that have subscribed to receive notifications for that operation.

We implement a high-performance RPC layer at the data plane using Apache Thrift [70] for interactions between clients and memory servers. While Thrift already provides low-overhead serialization/deserialization protocols, we add two key optimizations at the RPC layer. First, our server-side implementation employs asynchronous framed IO to multiplex multiple client sessions, permitting requests across different sessions to be processed in a non-blocking manner for lower latency and higher throughput. Second, while our client-side library is implemented in Python for compatibility with AWS Lambda, it employs thin Python wrappers around Thrift's C-libraries to minimize performance overheads.

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 Fig. 8); 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 [71] at block granularity and synchronously persisting data to external stores (e.g., S3) at address-prefix granularity.

2.7 Jiffy Programming Model

2.7.1 Map-Reduce Model

A Map-Reduce (MR) program [53] comprises map functions that process a series of 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 [53, 67, 72] 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 in a way that ensures values belonging to the same key are routed to the same worker.

MR on Jiffy executes map/reduce tasks as serverless tasks. A master process launches, tracks progress of, and handles failures for tasks across MR jobs. Jiffy stores intermediate KV pairs across multiple shuffle files, where shuffle files contain a partitioned subset of KV pairs collected from all map tasks. Since multiple map tasks can write to the same shuffle file, Jiffy's strong consistency semantics ensures correctness. The master process handles explicit lease renewals.

Jiffy Files. A Jiffy file is a collection of blocks, each storing a fixed-sized chunk of the file. The controller stores the mapping between blocks and file offset ranges managed by them at the metadata manager; this mapping is cached at clients accessing the file, and updated whenever the number of blocks allocated to the file is scaled in Jiffy. The getBlock operator forwards requests to different file blocks based on the offset range for the request. Files support sequential reads, and writes via append-only semantics. For random access, files support seek with arbitrary offsets. Jiffy uses the provided offset to identify the corresponding block and forwards subsequent read requests to it. Finally, since files are append-only, blocks can only be added to it (not removed), and do not require repartitioning when new blocks are added.

2.7.2 Dataflow and Streaming Dataflow Models

In the dataflow programming model, programmers provide DAGs to describe an application's communication patterns. DAG vertices correspond to computations, while data channels form directed edges between them. We use Dryad [54] as a reference dataflow execution engine, where channels can be files, shared memory FIFO queues, etc. The Dryad runtime schedules DAG vertices across multiple workers based on their dataflow dependencies. A vertex is scheduled when all its input

channels are ready: a file channel is ready if all its data items have been written, while a queue is ready if it has any data item. Streaming dataflow [55] employs a similar approach, except channels are continuous event streams.

Dataflow on Jiffy maps each DAG vertex to a serverless task, while a master process handles scheduling, fault tolerance, and lease renewals for Jiffy. We use Jiffy FIFO queues and files as data channels. Since queue-based channels are considered ready as long as some vertex is writing to it, Jiffy allows downstream tasks to efficiently detect if items produced by upstream tasks are available via notifications.

Jiffy Queues. A FIFO queue in Jiffy is a continuously growing linked list of blocks, where each block stores multiple data items, and a pointer to the next block in the list. The queue size can be upper-bounded (in number of items) by specifying a maxQueueLength. The controller only stores the head and the tail blocks in the queue's linked list, which the client caches and updates whenever blocks are added/removed. The queue supports enqueue/dequeue to add/remove items. The getBlock operator routes enqueue and dequeue operations to the current tail and head blocks in the linked list, respectively. While blocks can be both added and removed from a queue, queues do not need subsequent data repartitioning. Finally, the queue leverages Jiffy notifications to asynchronously detect when there is data in the queue to consume, or space in the queue to add more items, via subscriptions to enqueue and dequeue, respectively.

2.7.3 Piccolo

Piccolo [56] is a data-centric programming model that enables distributed compute machines to share mutable, distributed state. In Piccolo, kernel functions specify sequential application logic and share state with concurrent kernel functions through a KV interface, while centralized control functions manage and coordinate both the shared KV stores and the instances of kernel functions. Concurrent updates to the same key in the KV store are resolved using user-defined accumulators.

Piccolo on Jiffy runs kernel functions across serverless tasks, while control tasks are managed by a centralized master process. The shared state is distributed across Jiffy's KV-store data structures (detailed below). KV-stores can be created either per kernel function or shared across multiple functions, depending on the application requirements. The master process also handles periodic lease renewals for Jiffy KV-stores. Similar to Piccolo, Jiffy checkpoints KV-stores by flushing them

to an external store.

Jiffy KV-store. The Jiffy KV-store hashes each key to one of H hash slots in the range [0, H-1] (H=1024 by default). The KV-store shards key-value pairs across multiple Jiffy blocks, with each block responsible for one or more hash slots within this range. Each hash slot is entirely contained within a single block. The controller stores the mapping between the blocks and the hash slots they manage; this metadata is cached at the client and updated during resource scaling. Each block stores the key-value pairs that hash to its slots in a hash table, with Jiffy utilizing cuckoo hashing [73] to support highly concurrent KV operations. The KV-store supports typical get, put, and delete operations through implementations of readOp, writeOp, and deleteOp operators. The getBlock operator routes requests to the appropriate KV-store blocks based on key hashes.

Unlike files and queues, data in the KV-store must be repartitioned when a block is added or removed. When a block nears its capacity, Jiffy reassigns half of its hash slots to a new block, transfers the corresponding key-value pairs, and updates the block-to-hash-slot mapping at the controller. Similarly, when a block is nearly empty, its hash slots are merged with another block.

2.8 Applications and Evaluation

2.9 Related Work

2.10 Conclusion

Chapter 3

Operating System Layer

In the previous chapter we explore a design of memory management for disaggregated architecture in the service layer. However, integrating general application with an external memory service is challenging. In this chapter, we explore how to follow the class design of operating system, and leave the memory management functionality within the operating system. Transparency is an important aspect when considering migrating existing data center applications on disaggregated architecture. The operating system layer plays a crucial role in supporting the core functionality of a disaggregated architecture. This includes tasks like thread scheduling and data movement (paging). One of the key questions that arises is where the operating system should be situated within this architecture. There are two main options to consider:

Centralized OS Management. One approach is to place the operating system at a central point within the system, providing it with a global view. The advantage of this approach is that it maintains a well-defined operating system structure, requiring only minor modifications for application integration. However, ensuring that the central OS design doesn't introduce significant overhead is essential since the operating system typically lies on the critical path for applications, such as paging.

Disaggregation of OS Functions. An alternative approach involves the disaggregation of operating system functions across various resource blades, a concept explored in [**?**]. The rationale behind this approach is that many OS functionalities are closely intertwined with specific resources and remain largely independent of other system components. For instance, GPU driver functionality

can be situated within GPU resource pools rather than near compute or memory nodes. While this approach offers enhanced flexibility, it requires a substantial effort to overhaul the operating system. It may introduce synchronization overhead due to the inherently distributed nature of the system, necessitating additional coordination.

In the upcoming subsections, we present a hierarchical OS design, combining elements from the previously discussed options. Subsequently, we delve into our validation efforts concerning centralized and disaggregated OS functionality. Finally, we introduce prospective avenues for future work.

3.1 Hierarchical OS design

Rather than exclusively opting for one of these two approaches, we advocate for a hybrid OS design that integrates elements from both options mentioned earlier. Our observation suggests that operating system functionality can be classified into two distinct groups:

Non-disaggregated Functionalities. This category encompasses OS functionality that necessitates a holistic view of the entire system, including tasks like thread scheduling and memory management tasks such as memory address translation, protection, and paging. The operating system actively monitors the whole system, including available memory and compute resources, dynamically allocating computing and data resources to optimize system performance.

Disaggregated Functionalities. In contrast, this category comprises OS functions closely intertwined with specific resource types, including memory, SSD, or GPU drivers. In these contexts, it is more logical to position the functionality near the respective resource itself. Regarding memory management, this entails the implementation of memory access optimizations, such as enhancing the speed of irregular memory access. These optimization processes do not interact with other system components, obviating the need for a global view of the system.

3.2 In-Network Memory Management

3.2.1 Introduction

The current state of data center network bandwidth is rapidly approaching parity with intraserver resource interconnects, with projections indicating an imminent surpassing of this threshold. This dynamic shift has ignited considerable interest within both academic and industrial circles towards

memory disaggregation—a paradigm where compute and memory are physically decoupled into network-attached resource blades. This transformation promises to revolutionize resource utilization, hardware diversity, resource scalability, and fault tolerance compared to conventional data center architectures.

However, memory disaggregation presents formidable challenges, primarily revolving around three key requisites. Firstly, remote memory access demands low latency and high throughput, with previous studies targeting latency under 10 microseconds and bandwidth exceeding 100 Gbps per compute blade to minimize performance degradation in applications. Secondly, both memory and compute resources must exhibit elastic scalability, aligning with the essence of disaggregation. Lastly, seamless adoption and immediate deployment necessitate compatibility with unaltered applications.

Despite years of concerted research efforts directed towards enabling memory disaggregation, existing approaches have failed to concurrently meet all three requirements. Most strategies mandate application modifications due to alterations in hardware, programming models, or memory interfaces. Recent endeavors facilitating transparent access to disaggregated memory have encountered limitations on application compute elasticity—processes are confined to compute resources on a single blade to mitigate cache coherence traffic over the network, driven by performance apprehensions.

Introducing MIND, a pioneering memory management system tailored for rack-scale memory disaggregation, which effectively fulfills all three prerequisites for disaggregated memory. At the core of MIND lies a novel concept—embedding memory management logic and metadata within the network fabric. This innovative approach capitalizes on the insight that the network fabric in a disaggregated memory architecture essentially functions as a CPU-memory interconnect. In MIND, programmable network switches, strategically positioned for in-network processing, assume the mantle of Memory Management Units (MMUs), enabling a high-performance shared memory abstraction. Leveraging programmable hardware at line rate, MIND minimizes latency and bandwidth overheads.

However, the realization of in-network memory management necessitates navigating through the unique constraints imposed by programmable switch ASICs. These challenges include limited on-chip memory capacity, constraints on computational cycles per packet, and staged packet

processing pipelines spread across physically decoupled match-action stages.

To address the trifecta of requirements for memory disaggregation, MIND ingeniously maneuvers through these constraints and harnesses the capabilities of contemporary programmable switches to enable in-network memory management for disaggregated architectures. This is achieved through a systematic overhaul of traditional memory management mechanisms:

MIND adopts a globally shared virtual address space, partitioned across memory blades to minimize the volume of address translation entries stored in the on-chip memory of switch ASICs. Simultaneously, it implements a physical memory allocation mechanism that evenly distributes allocations across memory blades for optimal memory throughput.

MIND incorporates domain-based memory protection, inspired by capability-based schemes, facilitating fine-grained and flexible protection by dissociating the storage of memory permissions from address translation entries. Interestingly, this decoupling reduces on-chip memory overheads in switch ASICs.

MIND adapts directory-based MSI coherence to the in-network setting, leveraging network-centric hardware primitives like multicast in switch ASICs to efficiently realize its coherence protocol.

To mitigate the performance impact of coarse-grained cache directory tracking due to limited on-chip memory in switch ASICs, MIND introduces a novel Bounded Splitting algorithm that dynamically sizes memory regions to constrain both switch storage requirements and performance overheads stemming from false invalidations.

The MIND design is realized on a disaggregated cluster emulated using traditional servers connected by a programmable switch. Results demonstrate that MIND facilitates transparent resource elasticity for real-world workloads while matching or even surpassing the performance of prior memory disaggregation proposals. However, it's noted that workloads characterized by high read-write contention exhibit sub-linear scaling with additional threads due to the limitations of current hardware. Present x86 architectures hinder the implementation of relaxed consistency models commonly employed in shared memory systems, and the switch TCAM capacity nears saturation with cache directory entries for such workloads. Potential approaches for enhancing scalability with future advancements in switch ASIC and compute blade architectures are discussed.

3.2.2 Background and Motivation

This section motivates MIND. We discuss key enabling technologies, followed by challenges in realizing memory disaggregation goals using existing designs.

Assumptions: We focus on memory disaggregation at the rack-scale, where memory and compute blades are connected by a single programmable switch. We restrict our scope to partial memory disaggregation: while most of the memory is network-attached, CPU blades possess a small amount (few GBs) of local DRAM as cache.

2.1 Enabling Technologies We now briefly describe MIND's enabling technologies.

Programmable switches: In recent years, programmable switches have evolved along two well-coordinated directions: development of a flexible programming language for network switches and the design of switch hardware that can be programmed with it. These switches host an application-specific integrated circuit (ASIC), along with a general-purpose CPU with DRAM. The switch ASIC comprises ingress pipelines, a traffic manager, and egress pipelines, which process packets in that order. Programmability is facilitated through a programmable parser and match-action units in the ingress/egress pipelines.

The program defines how the parser parses packet headers to extract a set of fields, and multiple stages of match-action units process them. The general-purpose CPU is connected to the switch ASIC via a PCIe interface and serves two functions: (i) performing packet processing that cannot be performed in the ASIC due to resource constraints, and, (ii) hosting controller functions that compute network-wide policies and push them to the switch ASIC.

While this discussion focuses on switch ASICs with Reconfigurable Match Action Tables (RMTs), it is possible to realize MIND using FPGAs, custom ASICs, or even general-purpose CPUs. Each exposes different tradeoffs, but we adopt RMT switches due to their performance, availability, power, and cost efficiency.

DSM Designs: Traditionally, shared memory has been explored in the context of NUMA and distributed shared memory (DSM) architectures. In such designs, the virtual address space is partitioned across the various nodes, i.e., each partition has a home node that manages its metadata, e.g., the page table. Each node also has a cache to facilitate performance for frequently accessed memory blocks. We distinguish memory blocks from pages since caching granularities can be different from

memory access granularities.

With the copies of blocks potentially residing across multiple node caches, coherence protocols are required to ensure each node operates on the latest version of a block. In popular directory-based invalidation protocols like MSI (used in MIND), each memory block can be in one of three states: Modified (M), where a single node has exclusive read and write access to the block; Shared (S), where one or more caches have shared read-only access to the block; and Invalid (I), where the block is not present in any cache. A directory tracks the state of each block, along with the list of nodes that currently hold the block in their cache. The directory is typically partitioned across the various nodes, with each home node tracking directory entries for its own address space partition. Memory access for a block that is not local involves contacting the home node for the block, triggering a state transition and potential invalidation of the block across other nodes, followed by retrieving the block from the node that owns it.

While it is possible to realize more sophisticated coherence protocols, we restrict our focus to MSI in this work due to its simplicity.

As outlined earlier, extending the benefits of resource disaggregation to memory and making them widely applicable to cloud services demands (i) low-latency and high-throughput access to memory, and (ii) a transparent memory abstraction that supports elastic scaling of memory and compute resources without requiring modifications to existing applications. Unfortunately, prior designs for memory disaggregation expose a hard tradeoff between these two goals. Specifically, transparent elastic scaling of an application's compute resources necessitates a shared memory abstraction over the disaggregated memory pool, which imposes non-trivial performance overheads due to the cache-coherence required for both application data and memory management metadata. We now discuss why this tradeoff is fundamental to existing designs. We focus on page-based memory disaggregation designs here.

Transparent designs: While transparent distributed shared memories (DSMs) have been studied for several decades, their adaptation to disaggregated memory has not been explored. We consider two possible adaptations for the approach outlined earlier to understand their performance overheads and shed light on why they have remained unexplored thus far. The first is a compute-centric approach, where each compute blade owns a partition of the address space and manages the corresponding metadata, but the memory itself is disaggregated. A compute blade must now wait for

several sequential remote requests to be completed for every un-cached memory read or write, for example, to the remote home compute blade to trigger state transition for the block and invalidate relevant blades, and to fetch the memory block from the blade that currently owns the block.

An alternate memory-centric design that places metadata at corresponding home memory blades still suffers multiple sequential remote requests for a memory access as before, with the only difference being that the home node accesses are now directed to memory blades. While these overheads can be reduced by caching the metadata at compute blades, it necessitates coherence for the metadata as well, incurring additional design complexity and performance overheads.

Non-transparent designs: Due to the anticipated overheads of adapting DSM to memory disaggregation, existing proposals limit processes to a single compute blade, i.e., while compute blades cache data locally, different compute blades do not share memory to avoid sending coherence messages over the network. As such, these proposals achieve memory performance only by limiting transparent compute elasticity for an application to the resources available on a single compute blade, requiring application modifications if they wish to scale beyond a compute blade.

3.2.3 MIND Design

To break the tradeoff highlighted above, we place memory management in the network fabric for three reasons. First, the network fabric enjoys a central location in the disaggregated architecture. Therefore, placing memory management in the data access path between compute and memory resources obviates the need for metadata coherence. Second, modern network switches permit the implementation of such logic in integrated programmable ASICs. These ASICs are capable of executing at line rate even for multi-terabit traffic. In fact, many memory management functionalities have similar counterparts in networking, allowing us to leverage decades of innovation in network hardware and protocol design for disaggregated memory management.

Finally, placing the cache coherence logic and directory in the network switch permits the design of specialized in-network coherence protocols with reduced network latency and bandwidth overheads. Effective in-network memory management requires: (i) efficient storage by minimizing in-network metadata given the limited memory on the switch data plane; (ii) high memory throughput by load-balancing memory traffic across memory blades; and (iii) low access latency to shared memory via efficient cache coherence design that hides the network latency.

Next, we elicit three design principles followed by MIND to realize the above goals and provide an overview of its design.

MIND Design Principles

MIND adheres to three key principles to achieve the memory disaggregation goals outlined earlier:

P1: Decouple memory management functionalities to allow each to be optimized for its specific objectives. P2: Utilize a centralized control plane's global view of the disaggregated memory subsystem to compute optimal policies for each memory management functionality. P3: Leverage network-centric hardware primitives within the programmable switch ASIC to efficiently implement the policies determined by P2. MIND applies P1 by separating memory allocation from addressing, address translation from memory protection, and cache access and eviction from coherence protocol execution. P2 and P3 are employed to efficiently realize these objectives. Traditional server-based operating systems, however, are unable to take advantage of these principles due to their reliance on fixed-function hardware modules, such as the MMU and memory controller, which typically couple various memory management tasks (e.g., address translation and memory protection in page-table walkers) for reasons of complexity, performance, and power efficiency.

Overview

MIND provides a transparent virtual memory abstraction to applications, similar to traditional server-based OSes. However, unlike previous disaggregated memory designs, MIND places all memory management logic and metadata in the network, rather than on CPU or memory blades, or a separate global controller.

In MIND's design, CPU blades run user processes and threads and possess a small amount of local DRAM used as a cache. Memory allocations and deallocations from user processes are intercepted at the CPU blade and forwarded to the switch control plane. The control plane, which has a global view of the system, performs memory allocations, assigns permissions, and responds to user processes. All memory load/store operations are handled by the CPU blade's cache. This cache is virtually addressed and stores permissions to enforce memory protection. If a page is not cached locally, a page fault is triggered, causing the CPU blade to fetch the page from memory blades using RDMA requests, evicting other cached pages if necessary. If a memory access requires a coherence

state update (e.g., a store on a shared block), a page fault triggers the cache coherence logic at the switch.

MIND performs page-level remote accesses due to its page-fault-based design, although future CPU architectures may support more flexible access granularities. Since CPU blades do not store memory management metadata, the RDMA requests contain only virtual addresses, without any endpoint information for the memory blade holding the page. The switch data plane intercepts these requests, handles cache coherence by updating the cache directory, and performs cache invalidations on other CPU blades. It also ensures that the requesting process has the appropriate permissions. If no CPU blade cache holds the page, the data plane translates the virtual address to a physical one and forwards the request to the appropriate memory blade.

In this design, memory blades merely store the actual memory pages and serve RDMA requests for physical pages. Unlike earlier approaches that rely on RPC handlers and polling threads, MIND uses one-sided RDMA operations to eliminate the need for CPU cycles on disaggregated memory blades, moving towards true hardware resource disaggregation where memory blades do not need general-purpose CPUs. Placing memory management logic and metadata in the network enables simultaneous optimization for both memory performance and resource elasticity. We now explain how MIND optimizes for the goals of memory allocation and addressing, memory protection, and cache coherence, while adhering to the constraints of programmable switches. We also discuss how MIND handles failures.

4.1 Memory Allocation & Addressing Traditional virtual memory uses fixed-sized pages as basic units for translation and protection, which can lead to inefficiencies in storage due to memory fragmentation. Smaller pages reduce fragmentation but require more translation entries, and larger pages have the opposite effect. To address this, MIND decouples address translation from protection. MIND's translation is blade-based, while protection is virtual memory area (vma)-based.

Storage-efficient address translation: MIND avoids page-based protection and instead uses a single global virtual address space across all processes, allowing shared translation entries. MIND partitions the virtual address space across different memory blades, mapping each blade's portion to a contiguous physical address range. This approach reduces the storage needed for translation entries in the switch's data plane. The mapping is adjusted when memory blades are added, removed, or when memory is moved.

Balanced memory allocation & reduced fragmentation: The control plane tracks total memory allocation across blades and places new allocations on blades with the least allocation, achieving load balancing. Additionally, MIND minimizes fragmentation within each memory blade by using traditional virtual memory allocation schemes, resulting in virtual memory areas (vmas) that are non-overlapping, reducing fragmentation.

Isolation: MIND's global virtual address space does not compromise process isolation. The switch control plane intercepts all allocation requests and ensures that they do not overlap between processes. MIND's vma-based protection allows for flexible access control within a global virtual address space.

Support for static virtual addresses: MIND supports unmodified applications with static virtual addresses embedded in their binaries or OS optimizations like page migration. It maintains separate range-based address translations for static virtual addresses or migrated memory, ensuring correctness through longest-prefix matching in the switch's TCAM.

4.2 Memory Protection MIND decouples translation from protection by using a separate table to store memory protection entries in the data plane. Applications can assign access permissions to vmas of any size, and the protection table stores entries for these vmas. This flexible protection system allows MIND to efficiently manage memory protection with a relatively small number of entries.

Fine-grained, flexible memory protection: MIND introduces two abstractions: protection domains and permission classes. Protection domains define which entities can access a memory region, while permission classes specify the types of access allowed. MIND's control plane provides APIs that allow applications to assign protection domains and permission classes to vmas. These entries are stored in the protection table, and MIND efficiently supports this matching using TCAM-based range matches in the switch ASIC.

Optimizing for TCAM storage: MIND ensures storage efficiency by aligning virtual address allocations to power-of-two sizes, allowing regions to be represented using a single TCAM entry. Adjacent entries with the same protection domain and permission class are coalesced to further reduce storage requirements.

4.3 Caching & Cache Coherence In MIND, caches reside on compute blades, while the coherence directory and logic are located in the switch. This placement reduces latency for coherence

protocol execution. MIND addresses challenges in adapting traditional cache management to an in-network setting by decoupling cache and directory granularities and dynamically optimizing region sizes.

Decoupling cache access & directory entry granularities: MIND decouples cache access from directory entry granularity. Cache accesses and memory movements are performed at fine granularities (e.g., 4 KB pages), while directory entries are tracked at larger, variable-sized regions. Invalidation of a region triggers the invalidation of all dirty pages tracked by the CPU blade caches.

Storage & performance-efficient sizing of regions: MIND uses the global view of memory traffic at the switch control plane to dynamically adjust region sizes, balancing between performance (minimizing false invalidations) and directory storage efficiency.

4.4 Handling Failures MIND leverages prior work to handle CPU and memory blade failures. For switch failures, the control plane is consistently replicated at a backup switch, ensuring that data plane state can be reconstructed.

Communication failures: MIND uses ACKs and timeouts to detect packet losses. In case of a timeout during invalidation, the compute blade sends a reset message to the control plane, which flushes the data and removes the corresponding cache directory entry, preventing deadlocks during state transitions.

In-Network Memory Management

3.2.4 MIND Implementation

MIND Implementation MIND integrates with the Linux memory and process management system call APIs and splits its kernel components across CPU blades and the programmable switch. We will now describe these kernel components, along with the RDMA logic required for the memory blades.

6.1 CPU Blade MIND uses a partial disaggregation model, where CPU blades have a small amount of local DRAM that acts as a cache. In our prototype, traditional servers are used for the CPU blades, with no hardware modifications. We implemented MIND's CPU blade kernel components as modifications to the Linux 4.15 kernel, providing transparent access to disaggregated memory by modifying how vmas and processes are managed and how page faults are handled.

Managing vmas: The kernel module intercepts process heap allocation and deallocation requests, such as brk, mmap, and munmap, forwarding them to the control plane at the switch over a reliable TCP connection. The switch creates new vma entries and returns the corresponding values (e.g., the virtual address of the allocated vma), ensuring transparency for user applications. Error codes like ENOMEM are returned for errors, similar to standard Linux system calls.

Managing processes: The kernel module also intercepts and forwards process creation and termination requests, such as exec and exit, to the switch control plane, which maintains internal process representations (i.e., Linux's task_struct) and manages the mapping between compute blades and the processes they host. Threads across CPU blades are assigned the same PID if they belong to the same process, enabling them to share the same address space transparently through the memory protection and address translation rules installed at the switch. We place threads and processes across compute blades in a round-robin fashion without focusing on scheduling.

Page fault-driven access to remote memory: When a user application attempts to access a memory address not present in the CPU blade cache, a page fault handler is triggered. The CPU blade sends a one-sided RDMA read request to the switch with the virtual address and requested permission class (read or write). The page is registered to the NIC as the receiving buffer, eliminating the need for additional data copies. Once the page is received, the local memory structures are populated, and control is returned to the user. The CPU blade DRAM cache handles cache invalidations for coherence, tracking writable pages locally and flushing them when receiving invalidation requests.

This approach provides transparent access to disaggregated memory but restricts MIND to a stronger Total Store Order (TSO) memory consistency model. Weaker consistency models, such as Process Store Order (PSO), which allow asynchronous propagation of writes, are challenging to implement on traditional x86 and ARM architectures due to the inability to trigger page faults only on reads without also triggering them on writes. This limitation affects scalability for workloads with high read/write contention to shared memory regions.

6.2 Memory Blade MIND does not require any compute or data plane processing logic on memory blades, eliminating the need for general-purpose CPUs. In our prototype, memory blades are traditional Linux servers, so we use a kernel module to perform RDMA-specific initializations. When a memory blade comes online, its kernel registers physical memory addresses to the RDMA

NIC and reports them to the global controller. After this, one-sided RDMA requests from CPU blades are handled directly by the memory blade NIC without CPU involvement. Ideally, future memory blades could be fully implemented in hardware, without requiring a CPU, to reduce costs and simplify design.

6.3 Programmable Switch MIND's programmable switch is implemented on a 32-port EdgeCore Wedge switch with a 6.4 Tbps Tofino ASIC, an Intel Broadwell processor, 8 GB of RAM, and 128 GB of SSD storage. The general-purpose CPU hosts the MIND control program, handling process, memory, and cache directory management, while the ASIC performs address translation, memory protection, directory state transitions, and virtualizes RDMA connections between compute and memory blades.

Process & memory management: The control plane hosts a TCP server to handle system call intercepts from CPU blades and maintains traditional Linux data structures for process and memory management. Upon receiving a system call, the control plane updates these structures and responds with system call return values to maintain transparency.

Cache directory management: MIND reserves SRAM at the switch's data plane for directory entries, partitioned into fixed-size slots, one per memory region. The control plane maintains a free list of available slots and a hash table mapping base virtual addresses of cache regions to their corresponding directory entries in the SRAM. When a directory entry is created or a region is split, slots are allocated or deallocated as needed. Directory state transitions are handled across multiple match-action units (MAUs) due to limited compute capabilities in each unit, with state transitions split between them and recirculating the packet within the switch data plane as needed.

Virtualizing RDMA connections: MIND virtualizes RDMA connections between all possible CPU and memory blade pairs by transforming and redirecting RDMA requests and responses. Once a request's destination is identified through address translation or cache coherence, the switch updates the packet header fields (IP/MAC addresses and RDMA parameters) before forwarding the request to the correct memory blade.

3.2.5 Evaluation

3.2.6 Discussion and Conclusion

We start at a relatively modest scale, specifically within the context of rack-scale [**?**,**?**]. Our perspective aligns with placing the operating system functionality for non-disaggregated resources within the interconnect, which serves as the network infrastructure in a rack-scale system (or potentially utilizing CXL, as discussed in §**??**). The advantage of housing this functionality in the interconnect is it grants the system a global view, as every compute-memory operation must traverse the interconnect.

The network emerges as a compelling choice for an interconnect in memory disaggregation due to several key factors. First, the expansion of network bandwidth surpassing that of memory bandwidth [**?**] positions it as a prime candidate for serving as a disaggregation interconnect. Furthermore, advancements in programmable networking, exemplified by programmable switches [**?**, **?**, **?**, **?**], enable capabilities such as data storage (state-keeping) and processing at line-rate [**?**]. These capabilities empower the network to implement critical OS functionality effectively.

There are several essential requirements for memory management within a disaggregated architecture. Firstly, the interconnect operating system must operate without additional overhead, ensuring minimal latency and facilitating high-throughput access to remote memory. Additionally, given that programs may utilize various resources across compute and memory blades, the operating system should enable elastic scaling for both memory and computational resources. Another advantageous aspect of housing OS functionality within the interconnects is the ability to shield the application entirely from the OS logic, thereby promoting compatibility with unmodified applications.

To fulfill the three essential requirements, we have developed a system known as MIND [**?**], leveraging the capabilities of contemporary programmable switches to facilitate in-network memory management. Drawing inspiration from the similarity between memory address translation and network address lookups, we utilize the existing ingress/egress pipelines and Reconfigurable Match Action Tables (RMTs) [**?**] within programmable switches to implement address translation tables and protection entries. Additionally, we implement a directory-based MSI coherence protocol [**?**], as data may be accessed coherently by multiple compute nodes. These operations are performed at

line rate, ensuring low-latency, high-throughput memory access. It's worth noting that our implementation is confined to the interconnect (programmable switch) and the compute node OS kernel, allowing applications to run seamlessly on MIND.

Figure **??** illustrates the fundamental structure of the MIND system. Compute nodes house CPUs and a limited cache, while memory nodes exclusively contain memory resources. The programmable switch is situated atop the rack, with the control plane managing coarse-grained operations like memory allocation, permission assignment, and memory coherence directory management. Meanwhile, the data plane handles memory address translation, protection, and coherence lookup at line rate.

The dataflow(Figure **??**) of memory access begins with a load/store instruction from the compute node CPU. When the compute node OS kernel detects that the required data isn't present on the node, it triggers a page fault and issues a network request to the switch for permission updates and data retrieval. This request traverses the switch's data plane, fetching the required data from the memory node. Simultaneously, the switch invalidates existing data from other compute nodes if the source node requests exclusive access.

We've faced two main challenges with programmable switch ASICs: limited on-chip memory and restricted computational power. The few megabytes of memory on switch ASICs are inadequate for traditional page tables managing terabytes of disaggregated memory. Moreover, the ASICs' computational constraints, necessary for maintaining line-rate processing, are evident in complex tasks like cache coherence. To counter these issues, we've separated memory addressing and protection to save hardware space. Additionally, we've utilized unique switch primitives like multicast operations to navigate computational limitations effectively.

3.3 Near Memory Processing

3.4 Introduction

Driven by increasing demands for memory capacity and bandwidth [**?**, **?**, **?**, **?**, **?**, **?**, **?**], poor scaling [**?**, **?**, **?**] and resource inefficiency [**?**, **?**] of DRAM, and improvements in Ethernet-based network speeds [**?**, **?**], recent years have seen significant efforts towards memory disaggregation [**?**, **?**, **?**, **?**, **?**]. Rather than scaling up a server's DRAM capacity and bandwidth, such proposals advocate disaggregating much of the memory over the network. The result is a set of CPU nodes equipped

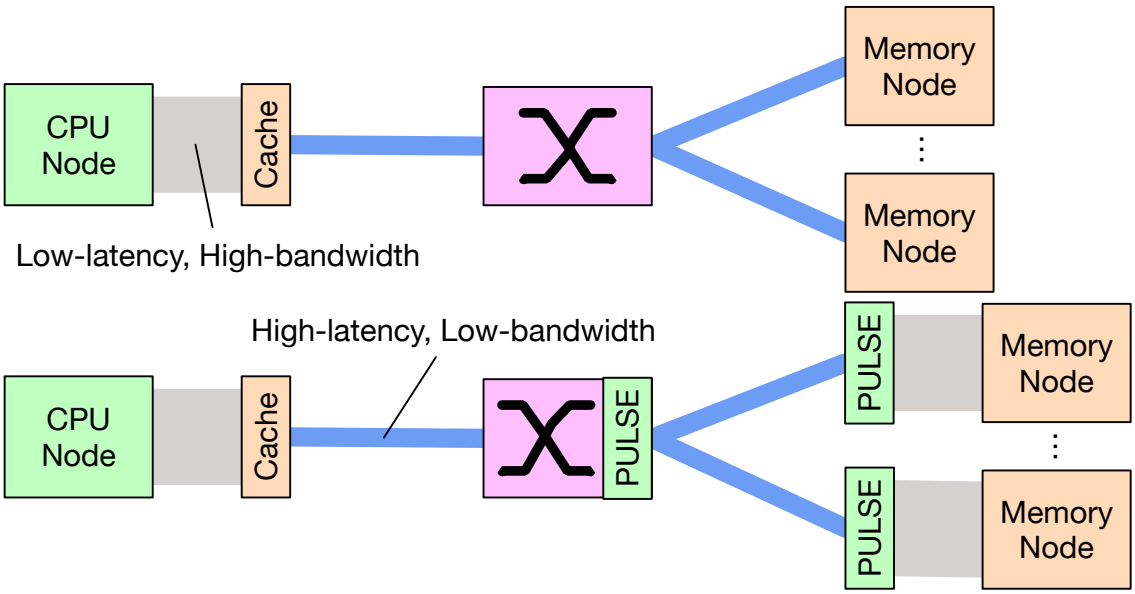
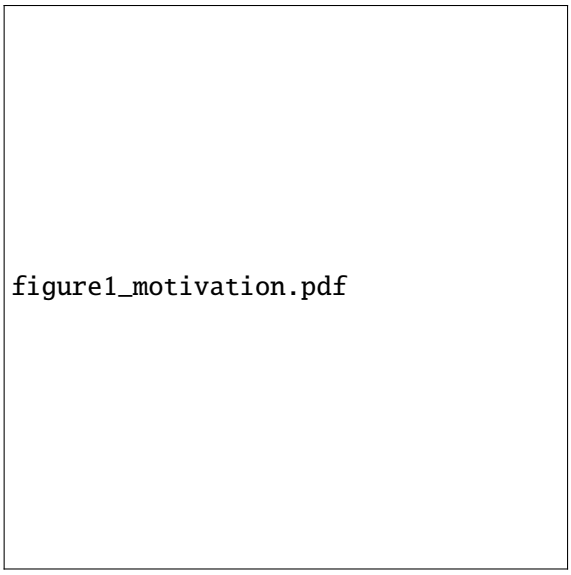


Fig. 3.1: **Need for accelerating pointer traversals.** *(top)* The performance of pointer traversals in disaggregated architectures is bottlenecked by slow memory interconnect. *(bottom)* Just as caches offer limited but fast caches near CPUs, we argue that memory needs a counterpart for traversal-heavy workloads: a lightweight but fast accelerator for cache-unfriendly pointer traversals.

with a small amount of DRAM used as cache[1], accessing memory across a set of network-attached memory nodes with large DRAM pools (Fig. **??** (top)). With allocation flexibility across CPU and memory nodes, disaggregation enables high utilization and elasticity.

Despite drastic improvements in recent years, the limited bandwidth and latency to network-attached memory remain a hurdle in adopting disaggregated memory, with speed-of-light constraints making it impossible to improve network latency beyond a point. Even with near-terabit links and hardware-assisted protocols like RDMA [**?**], remote memory accesses are an order of magnitude slower than local memory accesses [**?**]. Emerging CXL interconnects [**?**] share a similar trend — around 300 ns of CXL memory latency compared to 10–20 ns of L3 cache latency [**?**]. Although efficient caching strategies at the CPU node can reduce average memory access latency and volume of network traffic to remote memory, the benefit of such strategies is limited by data locality and the size of the cache on the CPU node. In many cases, remote memory accesses are unavoidable, especially for applications that rely on efficient in-memory pointer traversals on linked data structures, such as lookups on index structures [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**] in databases and key-value stores, and traversals in graph analytics [**?**, **?**, **?**, **?**] (Fig. **??**, §**??**).

1. Not to be confused with die-stacked hardware DRAM caches [**?**, **?**, **?**].



(a) Our empirical analysis

(b) % of distributed traversals

(c) CDF of distributed traversals

Fig. 3.2: **Time cloud applications spend in pointer traversals.** See §**??** for details.

Similar to how CPUs have small but fast memory (i.e., caches) for quick access to popular data, we argue that memory nodes should also include lightweight but fast processing units with high-bandwidth, low-latency access to memory to speed up pointer-traversals (Fig. **??** (bottom)). Moreover, the interconnect should facilitate efficient and scalable distributed traversals for deployments with multiple memory nodes that cater to large-scale linked data structures. Prior works have explored systems and API designs for such processing units under multiple settings, ranging from near-memory processing and processing-in-memory approaches [**?**, **?**] for single-server architectures, to the use of CPUs [**?**, **?**, **?**, **?**, **?**, **?**] or FPGAs [**?**, **?**] near remote/disaggregated memory, but have several key shortcomings.

Specifically, existing approaches are limited in scale and expose a three-way tradeoff between expressiveness, energy efficiency, and performance. First, and perhaps most crucially, none of the

existing approaches can accelerate pointer traversals that span *multiple* network-attached memory nodes. This limits memory utilization and elasticity since applications must confine their data to a single memory node to accelerate pointer traversals. Their inability to support distributed pointer traversals stems from complex management of address translation state that is required to identify if a traversal can occur locally or must be re-routed to a different memory node (§**??**). Second, existing single-node approaches use full-fledged CPUs for expressive and performant execution of pointer-traversals [**?**, **?**, **?**, **?**]. However, coupling large amounts of processing capacity with memory — which has utility in reducing data movement in PIM architectures [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**] — 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 [**?**, **?**] 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 (§**??**, [**?**]). Lastly, FPGA-based [**?**, **?**, **?**] and ASIC-based [**?**, **?**] approaches achieve performance and energy efficiency by hard-wiring pointer traversal logic for specific data structures, limiting their expressiveness.

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

In particular, CHASE enables transparent and efficient execution of pointer traversals for our iterator abstraction via a novel accelerator that employs a *disaggregated* architecture to decouple logic and memory pipelines, exploiting the inherently sequential nature of compute and memory

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

accesses in iterator execution (§**??**). This permits high utilization by provisioning more memory and fewer logic pipelines to cater to memory-centric pointer traversal workloads. A scheduler breaks pointer traversal logic from multiple concurrent workloads across the two sets of pipelines and employs a novel multiplexing strategy to maximize their utilization. While our implementation leverages an FPGA-based SmartNIC due to the high cost and complexity of ASIC fabrication, our ultimate vision is an ASIC-based realization for improved performance and energy efficiency.

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

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

Chapter 4

Hardware Layer

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 [**?**] from Nvidia and Compute Express Link (CXL) [**?**] 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 [**?**, **?**, **?**], 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 [**?**, **?**], 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) [**?**], 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 [**?**] 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.

4.2 Introduction

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 [**?**, **?**, **?**]. 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 [**?**, **?**].

To meet such urgent demands, Compute Express Link (CXL) [**?**, **?**, **?**, **?**] is introduced as a groundbreaking interconnect technology. CXL promises significant expansion of memory capacity

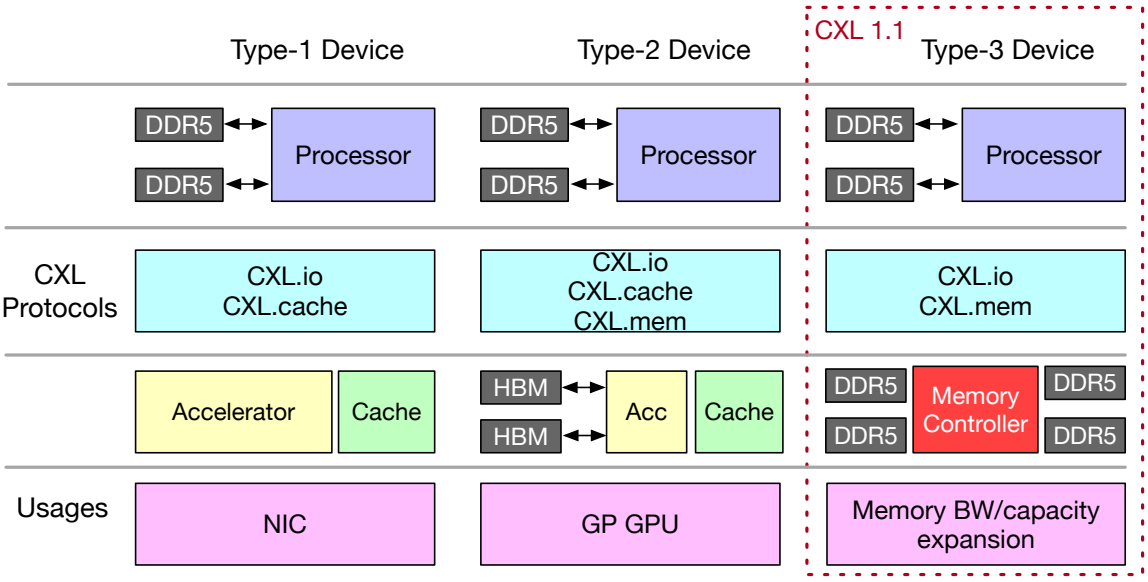


Fig. 4.1: **CXL Overview.** In this study, we focus on commercial CXL 1.1 Type-3 devices, leveraging CXL.io and CXL.mem protocols for memory expansion in single-server environments.

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 [**?**] 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 [**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**], 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 [**?**,**?**] or using FPGA-based setups [**?**,**?**]. Although a limited number of studies have begun to assess the raw performance of ASIC-based CXL hardware [**?**, **?**], 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 [**?**], a critical gap remains in understanding

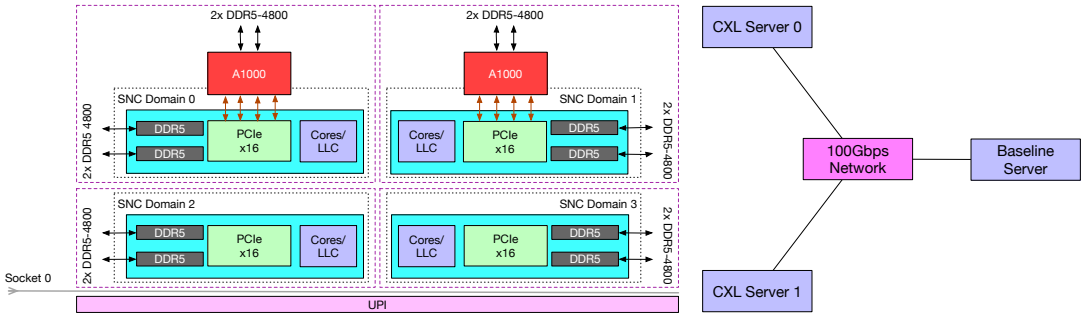
the cost-effectiveness of migrating particular types of applications or services to memory expansions facilitated by CXL. (3) Given the restricted availability of CXL ASIC hardware, the research community faces a notable scarcity of open-source empirical data. This limitation hinders efforts to fully 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 [**?**, **?**], **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 (§**??**) 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. §**??** introduces basic information of CXL and environment setup for the evaluations. §**??** presents basic performance characteristic of CXL memory expansion. §**??** and §**??** presents findings and suggestions of using CXL as the expansion of memory capacity and bandwidth on data center workloads. §**??** provides a detailed analysis on the potential cost benefits brought by CXL. §**??** discusses how our insights are applicable to future generations of CXL. §**??** describes related work, and §**??** concludes the paper.



(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(§**??**) is enabled only for the raw performance benchmarks(§**??**) and bandwidth-bound benchmarks(§**??**), 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.

4.3 Background and Methodology

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

4.3.1 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 [**?**], 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. **??**. *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 [**?**]. In this paper, our focus is on commercially available CXL 1.1 Type-3 devices, specifically addressing single-host memory expansion.

4.3.2 Hardware Support for CXL

Recent announcements have introduced CXL 1.1 support for Intel Sapphire Rapids processors (SPR) [**?**] and AMD Zen 4 EPYC "Genoa" and "Bergamo" processors [**?**]. While commercial CXL memory modules are provided by vendors such as Asteralabs [**?**], Montage [**?**], Micron [**?**], and Samsung [**?**], 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 [**?**, **?**] and FPGA implementations [**?**, **?**], 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 [**?**, **?**] 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 [**?**], as shown in previous research [**?**].

FPGA-based implementation. Intel and other hardware vendors use FPGA hardware to implement CXL protocols [**?**], 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 [**?**]. FPGAs prioritize flexibility over performance and are suitable for early-stage CXL memory validation but not production deployment. Intel's recent evaluation [**?**] 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 §**??**.

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 [**?**] 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.

4.3.3 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 [**?**] 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 [**?**], 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 §**??**, 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:

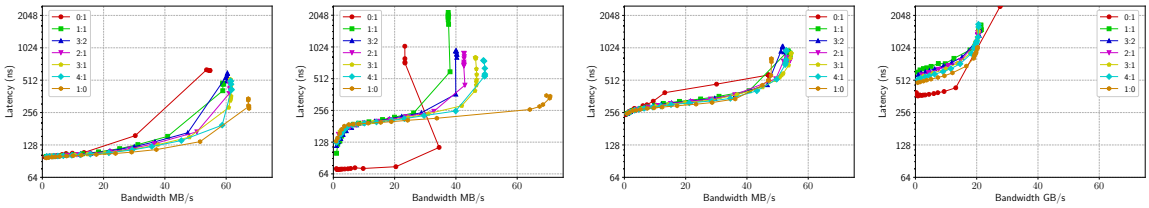
1. The *NUMA-balancing* patch [**?**] uses a latency-aware page migration strategy, focusing on promoting recently accessed pages (MRU). It scans NUMA balancing page tables and hints page faults. However, it may not accurately identify high-demand pages due to extended scanning intervals, potentially 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 [**?**] share a similar concept with optimizations and are being considered for integration into the Linux Kernel [**?**]. 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.

4.3.4 Experimental Platform Description

The evaluation testbed, as illustrated in Fig. **??**, 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 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.



(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 (§**??**). 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.

4.4 CXL 1.1 Performance Characteristics

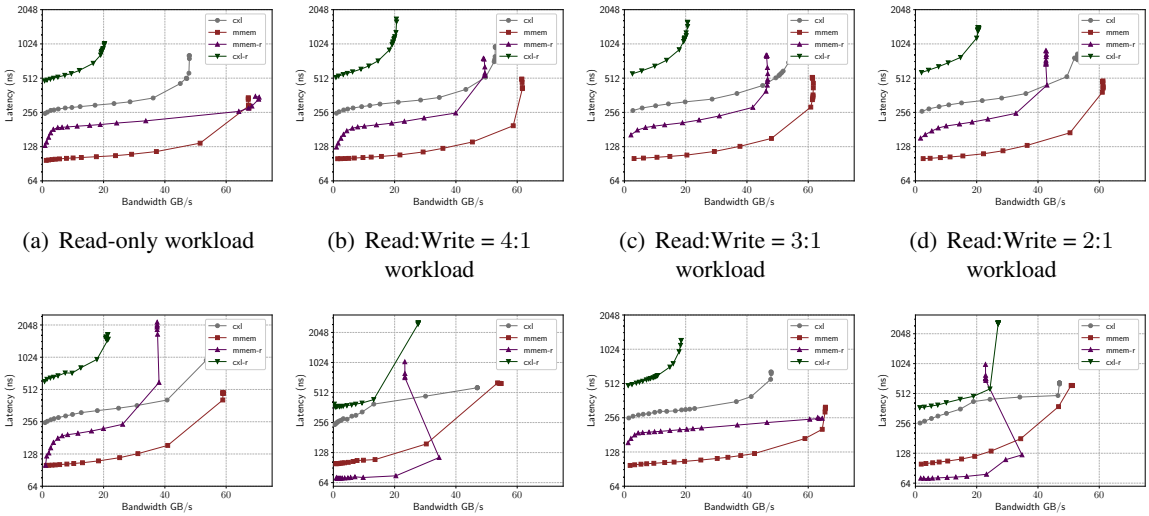
In this section, we assess the performance of the CXL memory expander and compare it directly with main memory, which we designate as **MMEM** for clarity against CXL memory. We analyze workload patterns and evaluate performance differences between local and remote socket scenarios.

4.4.1 Experimental Configuration

For each dual-channel A1000 ASIC CXL memory expander [**?**], 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) [**?**] 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 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. **??**), 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 [**?**]. 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)



(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.

to examine loaded-latency for various read-write workloads, adopting a 64-byte access size same as prior work [**?**]. 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?

4.4.2 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 **??** 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 [**?**, **?**]. Interestingly, latency starts to significantly increase at 75%-83% of bandwidth utilization, surpassing prior estimates of 60% from earlier studies [**?**].

Figure **??** 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. **??** 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. **??** 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. **??**. 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) [**?**] 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 [**?**].

4.4.3 Different Read-Write Ratios & Access Pattern

Fig. **??**-**??** 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. **??** and **??** 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.

4.4.4 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

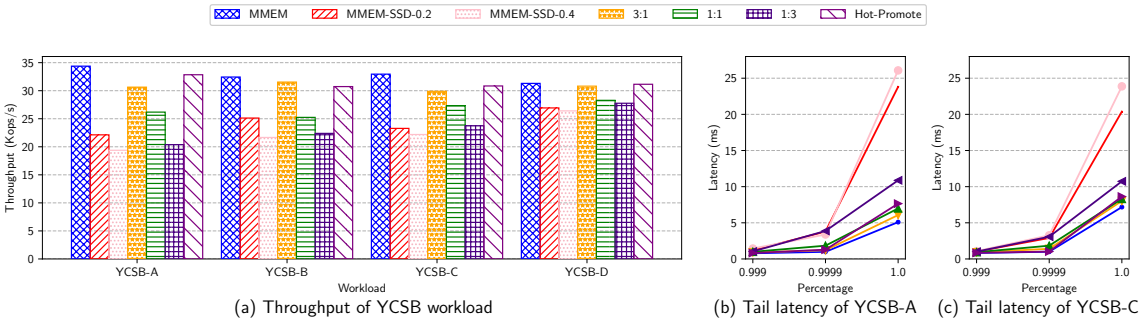
attainable when accessing CXL memory across sockets could approximate the bandwidth seen when accessing MMEM across sockets.

Bandwidth Contention Previous research [**?**, **?**] 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 [**?**]. 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' [**?**, **?**, **?**], 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 (§**??**).

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

4.5 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



(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 [**?**].

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.

4.5.1 In-memory key-value stores

Redis [**?**] 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 [**?**] in data centers. Google Cloud suggests keeping memory usage below 80% [**?**], whereas other sources recommend a limit of 75% [**?**].

Due to the substantial infrastructure costs for memory-only deployment, Redis Enterprise [**?**] is the commercial variant extensively supported by leading cloud platforms (e.g., AWS, Google Cloud, or Azure). It introduces "Auto Tiering" [**?**] 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.

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 §**??**.

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

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 **??**. 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.

Analysis.

Fig. **??** 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.

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.

4.5.2 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 [**?**]. Take Spark [**?**], one of the common Big Data platforms, as an example: A typical query requires shuffling data from multiple tables for processing in the next stage. Operations like

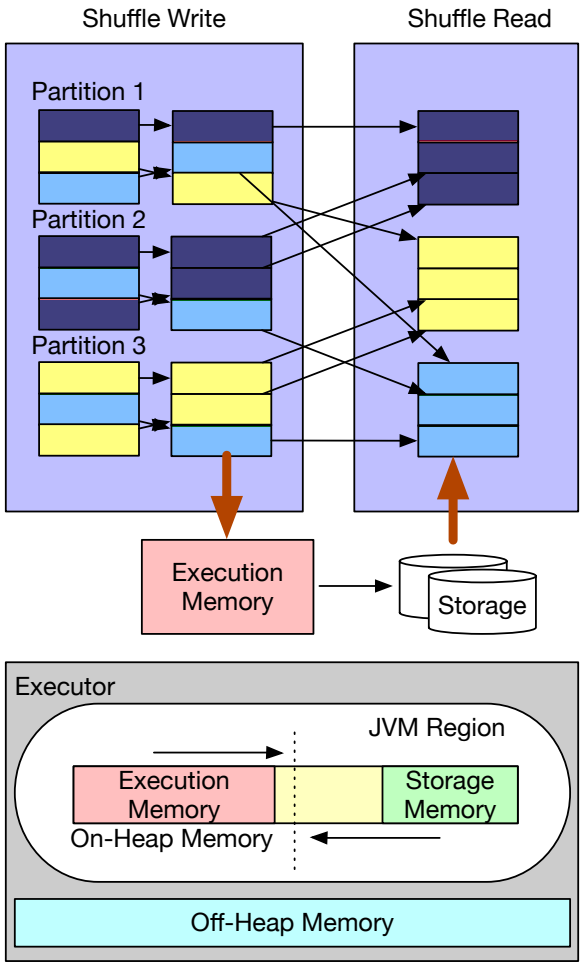
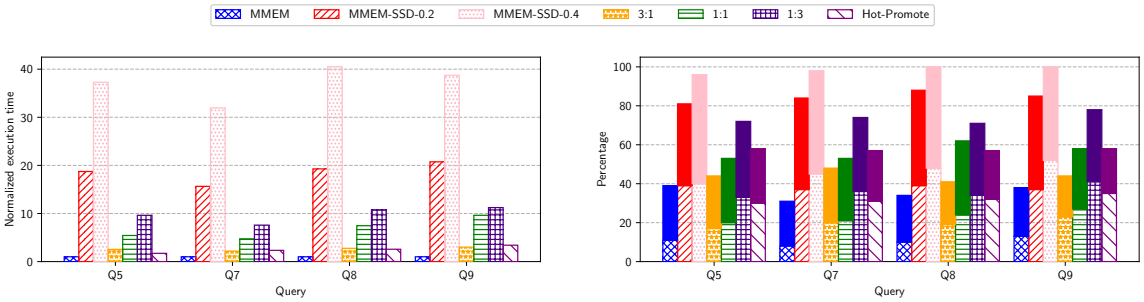


Fig. 4.6: **Spark memory layout and shuffle spill.** Each Spark executor possesses a fixed-size On-Heap memory, which is dynamically divided between execution and storage memory. If there is insufficient memory during shuffle operations, the Spark executor will spill the data to the disk.

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 [**?**]. During the shuffling process(Fig. **??**), 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.



(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.

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 [**?**] 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 **??** 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 (§**??**).

We chose four specific queries ($Q5$, $Q7$, $Q8$, and $Q9$) from the TPC-H benchmark [**?**], recognized for their intensive data shuffling demands from prior studies [**?**], 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 **??** 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 **??**(b) illustrates that shuffling overshadows the total execution time due to the intensification of data spill issues.

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

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

Table 4.2: **Intel Processor Series.**

issues highlighted in prior research [**?**].

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 §**??**. 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.

4.5.3 Spare Cores for Virtual Machine

One widely-used application within Infrastructure-as-a-Service (IAAS) is Elastic Computing [**?**]. 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 [**?**, **?**]). 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 **??**). 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 [**?**]. 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 (§**??**). 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.

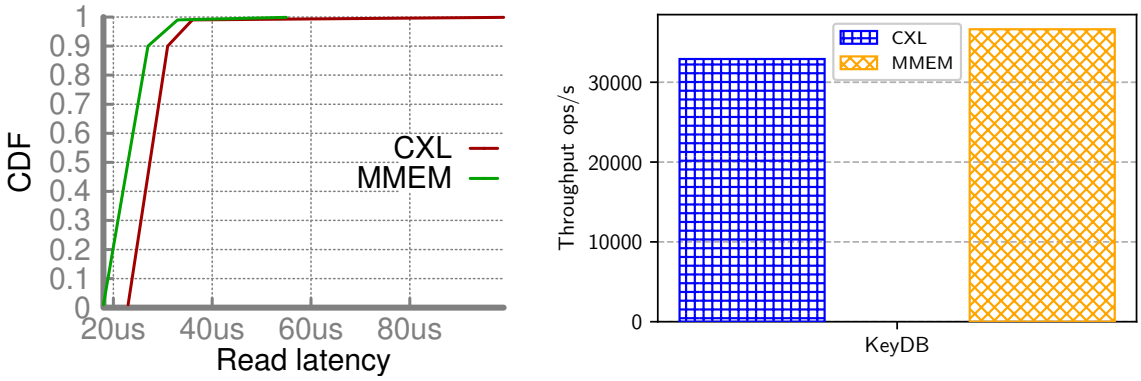
Analysis.

The CDF of read latency (Fig. **??**) 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 §**??**. 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. **??**.

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



(a) CDF of KeyDB YCSB-C

(b) Throughput of KeyDB YCSB-C

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

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 [**?**] 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.

4.6 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 [**?**] 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

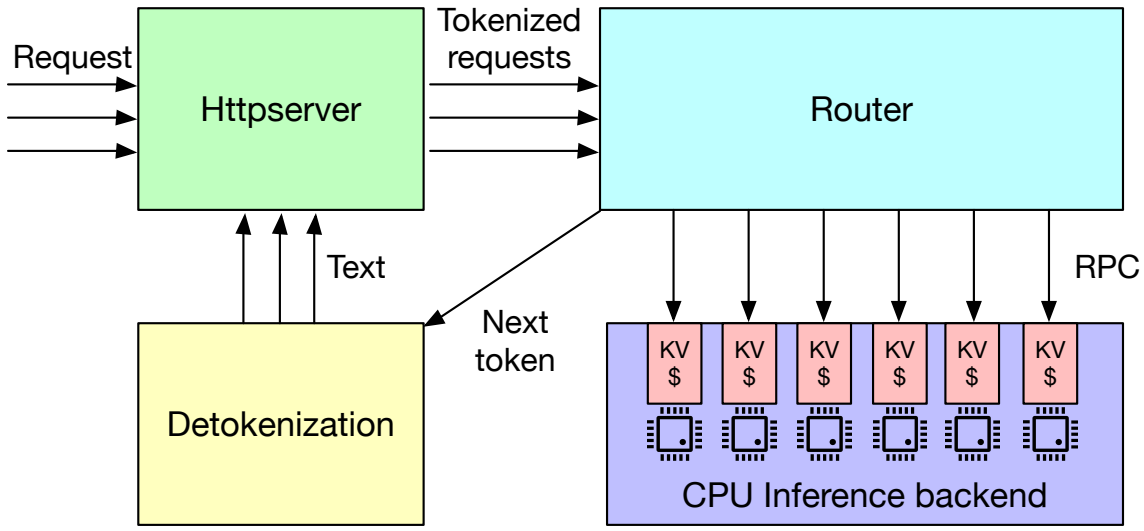
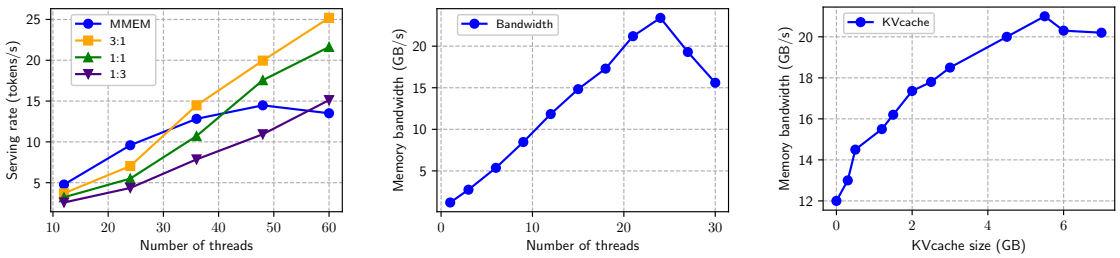


Fig. 4.9: **LLM inference framework.** The Httpserver receive requests and forward the tokenized requests to the CPU inference backend. The CPU inference backend serves the requests and reply the next token.

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.



(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.**

LLM Inference Framework. Mainstream Large Language Model (LLM) inference frameworks, such as vLLM [**?**] and LightLLM [**?**], do not support CPU inference. Recently, Intel introduced an LLM model named Q8chat [**?**], 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 [**?**] by replacing the backend with a CPU inference backend. Figure **??** 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 [**?**], 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 [**?**] 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 [**?**] 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 [**?**].

4.6.1 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 (§**??**). We examine three distinct interleaving policies (3:1, 1:1, 1:3), detailed in Table **??**. 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 [**?**], an advancement of the LLaMA 7B model, requiring 4.1GB of memory. The workload, derived from the LightLLM framework [**?**], 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).

4.6.2 Analysis

Fig. **??** 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 (§ **??**). Fig. **??** charts the memory bandwidth utilization, as measured by the Intel Performance Counter Monitor (PCM) [**?**], 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 §**??**, 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. **??** 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.

4.6.3 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%), exist-

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.**

ing page migration policy(§**??**) 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.

4.7 Cost Implications

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

Chapter 5

Future Work

Appendix A

Appendix

If you need an appendix, it will go here.