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 increasing demand for scalable and efficient data center architectures has given rise to the concept of resource disaggregation [1, 2]. This contemporary paradigm represents a significant departure from traditional monolithic server architectures. In conventional setups, servers typically come pre-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 [3], CXL [4], and others. This separation fosters more flexible resource allocation and utilization.

Within the broader context of resource disaggregation in modern data center architectures, memory disaggregation [5–10] plays a pivotal and foundational role. Memory often serves as a bottleneck in traditional monolithic server configurations, limiting the scalability and adaptability of applications. Data centers can achieve increased efficiency, scalability, and adaptability by decoupling memory resources from compute and storage elements and presenting them as pooled, disaggregated resources. Memory-intensive workloads can access the memory they require on demand without being constrained by the limitations of individual servers. Memory disaggregation serves as the initial step toward unlocking the full potential of resource disaggregation, enabling data centers to allocate and utilize resources based on dynamic application needs efficiently. This ultimately leads to improved performance and resource utilization.

While resource disaggregation offers numerous advantages, transitioning existing applications to a disaggregated architecture is far from straightforward. Recent research efforts have explored various approaches to tackle this challenge. Some have focused on adapting applications to optimize their utilization of disaggregated memory [11, 12], while others have aimed to transparently port applications and shift the responsibility of mitigating the performance penalty caused by the mismatch between disaggregated architecture and software interfaces to the service or operating system layer [1, 2, 13, 14].

The fundamental challenge is the mismatch between the existing cloud stack for monolithic architecture and what is required for disaggregated architecture(Figure **??**). The current cloud stack and hardware stack lack awareness of the unique characteristics of disaggregated memory. There are different requirements and challenges of 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 [1] in the architecture or disaggregating its functions across different resource blades [2].

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 [15] and addressing applications with irregular memory access patterns [16]. 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) [4] 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.

1.1 Thesis Overview

In this dissertation, we attempt to take a top-down approach and explore the solutions for each layer of disaggregated memory architectures. We focus on the challenges of three layers of memory management.

1.1.1 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.

1.1.2 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.

1.1.3 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.

1.2 Outline and Previously Published Work

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

Chapter 2 revises material from [15]. Chapter 3 revises material from [1] and [16]. Finally, Chapter 4 revises material from [17].

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 applications experiencing fluctuating memory demands [15]. 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 [18] 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 [15]. 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 [18–21], 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 [22, 23]. 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 [22], 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 [15], 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 [2]. 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 [24, 25]. 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 [26] positions it as a prime candidate for serving as a disaggregation interconnect. Furthermore, advancements in programmable networking, exemplified by programmable switches [27–30], enable capabilities such as data storage (state-keeping) and processing at line-rate [31]. 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 [1], 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) [32] within programmable switches to implement address translation tables and protection entries. Additionally, we implement a directory-based MSI coherence protocol [33], 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

Remote memory accesses via interconnects are considerably slower compared to local memory accesses. This is particularly true for applications dependent on efficient in-memory pointer traversals within linked data structures. Near Memory Processing (NMP) emerges as an effective solution to this challenge, also serving as a promising candidate for disaggregated OS functionality. This is due to its close integration with memory nodes. In this context, we have identified and summarized the key requirements for a near-memory processor, considering its specific computational needs.

Controlled expressiveness. The NMP interface must balance generality and specificity. It should be versatile enough to accommodate a range of applications, particularly those with irregular access patterns. However, it must also avoid offloading tasks that do not benefit from such a process, such as compute-intensive applications. The focus in near-memory offloading should be on memory-centric, rather than compute-centric, logic. For compute-centric workloads, transferring data to the corresponding compute node for processing is more logical.

Energy Efficiency. An NMP accelerator must be energy-efficient, incorporating only the necessary amount of computing power. The memory node must not house a full-scale CPU to enhance resource utilization. Instead, it should feature a custom ASIC designed solely for managing irregular data access.

Scalability. Scalability is key for NMP, particularly in supporting pointer traversal, as data may be distributed across multiple memory nodes. Without a mechanism for seamless traversal through various nodes, applications may need to revert to the compute node to determine the location of subsequent data. This limitation can significantly hinder efficiency.

While previous studies [12, 34–36] have extensively explored near-memory processing in the context of far-memory, they do not simultaneously meet the criteria of expressiveness, energy efficiency, and performance due to inherent trade-offs. Solutions utilizing RPC and fully-equipped CPUs [12, 37] offer general-purpose processing with commendable performance but lack energy efficiency. Conversely, dedicated hardware solutions [34, 36] optimize performance for specific applications but fail to support a broader range of applications. Alternatives employing wimpy cores for near-memory processing [35] fall short in performance and energy efficiency, mainly due to extended execution times.

To address the three fundamental requirements, we developed a novel OS-level NMP accelerator framework, CHASE [16]. Our framework introduces an iterator-based interface that aligns well with the commonly used iterators in C++ and Java data structures. This design ensures broad applicability across various applications while focusing primarily on memory-centric processing. Additionally, we have innovatively designed a memory-compute decoupled architecture that not only achieves energy efficiency but also fully utilizes memory bandwidth. By integrating the CHASE iterator-based interface with a programmable switch's global view, we facilitate distributed continuation, enhancing the efficiency of pointer traversal workloads.

As depicted in Figure **??**, the CHASE framework features compute nodes equipped with CPUs to handle applications that require irregular data access patterns. Notably, application developers can integrate with CHASE without modifying their existing code. This ease of integration is possible using standard data structure libraries like STL or Boost. Developers can leverage the framework seamlessly by linking their applications with the CHASE-modified libraries, which retain the same programming interface. The CHASE compiler plays a pivotal role by translating the iterator interface into the CHASE Instruction Set Architecture (ISA), a specialized subset of the RISC ISA. Subsequently, the offload engine encapsulates these requests into UDP packets and transmits them via the network interconnect. Atop each rack sits a programmable switch, essential in directing requests to the appropriate memory node. This process mirrors the approach outlined in Section §**??**. Each memory node contains a CHASE near-memory accelerator, processing the iterator microcode and returning responses to the compute nodes.

Distributed Continuation. A key feature of CHASE is its distributed continuation mechanism. When a memory node identifies that the following required pointer is not within its storage, it returns the request to the switch. This return packet includes both the original iterator microcode and an updated pointer. The programmable switch, equipped with range-based address translation capabilities, efficiently forwards this request to the next relevant memory node. Thus, the compute node receives the final result only after the complete traversal is executed, ensuring efficient data processing across distributed systems.

3.3.1 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 with a small amount of DRAM used as cache, accessing memory across a set of network-attached memory nodes with large DRAM pools. With allocation flexibility across CPU and memory nodes, disaggregation enables high utilization and elasticity.

Despite 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 significantly slower than local memory accesses. Emerging CXL interconnects share a similar trend — remote memory accesses incur much higher latency compared to local memory accesses. Although efficient caching strategies at the CPU node can reduce average memory access latency and network traffic volume 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.

Similar to how CPUs have small but fast memory (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. 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 these approaches have several key shortcomings.

Existing approaches are limited in scale and expose a tradeoff between expressiveness, energy efficiency, and performance. First, 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 the 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 leads to poor utilization of compute resources and poor energy efficiency. Approaches that use wimpy processors at SmartNICs retain expressiveness but suffer from limited processing speeds, which curtails their performance and ultimately leads to lower energy efficiency. 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 a distributed pointer-traversal framework for rack-scale disaggregated memory, to meet the needs for expressiveness, energy efficiency, and performance via a principled redesign of near-memory processing for disaggregated memory. Central to the design is an expressive iterator interface that serves as a unifying abstraction across most pointer traversals in linked data structures used in key-value stores, databases, and big-data analytics. The use of this abstraction makes the framework immediately useful in a large family of traversal-heavy real-world use cases and 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.

The framework enables transparent and efficient execution of pointer traversals for our iterator abstraction via a novel accelerator that employs a disaggregated architecture to decouple logic and memory pipelines, exploiting the inherently sequential nature of compute and memory accesses in iterator execution. 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 multiplexing strategy to maximize their utilization. While our implementation leverages an FPGA-based SmartNIC, 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. The framework 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 the framework on a disaggregated rack of commodity servers, SmartNICs, and a programmable switch with full-system effects. None of the hardware or software changes are invasive or overly complex, ensuring deployability. Our evaluation of end-to-end real-world workloads shows that the framework outperforms disaggregated caching systems with significantly lower latency and higher throughput. Moreover, our power analysis shows that the framework consumes considerably less energy than RPC-based schemes.

3.3.2 Motivation

Need for Accelerating Pointer Traversals Memory-intensive applications often require traversing linked structures like lists, hash tables, trees, and graphs. While disaggregated architectures provide large memory pools across network-attached memory nodes, traversing pointers over the network remains slow. Recent proposals alleviate this slowdown by using the DRAM at CPU nodes to cache "hot" data, but such caches often perform poorly for pointer traversals, as we show next.

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

Distributed traversals: As the number of applications and their working set sizes grow, disaggregated architectures must allocate memory across multiple memory nodes to keep up. These approaches tend to use smaller allocation granularities to achieve better load balancing and high memory utilization. Unfortunately, finer-grained allocations may cause an application's linked structures to fragment across multiple network-attached memory nodes, necessitating many distributed traversals. This increases the volume of cross-node traffic and impacts performance, especially for applications where random or time-ordered data insertion spreads data across memory nodes.

Shortcomings of Prior Approaches No prior work achieves all the required properties for pointer traversals on disaggregated memory: distributed execution, expressiveness, energy efficiency, and performance. We focus on network-attached memory, although similar issues arise in near-memory processing.

No support for distributed execution: Distributed pointer traversals are essential for efficiently accessing large pools of network-attached memory nodes. However, prior work does not support efficient multi-node pointer traversals. Consequently, applications must confine their data to a single node for efficient traversals, leading to tradeoffs between performance and scalability. Specialized data structures co-designed with partitioning and allocation policies to reduce distributed pointer traversals complement our work but still require efficient distributed traversal mechanisms when their optimizations are not applicable.

Poor utilization/power-efficiency in CPUs: Many previous works have explored remote procedure call (RPC) interfaces to offload computation to CPUs on memory nodes. While CPUs are versatile enough to support general-purpose computations, they are often overkill for pointer traversal workloads in disaggregated architectures. These workloads are typically memory-intensive and constrained by memory bandwidth rather than CPU cycles. As a result, the CPUs on memory nodes are likely to be underutilized, leading to wasted energy. Using CPUs for pointer traversal workloads can nullify the benefits of disaggregation by coupling compute and memory resources inefficiently.

Limited expressiveness in FPGA/ASIC accelerators: FPGA-based and ASIC-based approaches at memory nodes offer performance and energy efficiency but are limited in expressiveness. FPGA approaches typically perform on-path data processing for specific data structures, limiting their flexibility. While some FPGA approaches aim to be more expressive by supporting RPCs, they are constrained by the need to pre-compile RPC logic, which physically consumes FPGA resources and limits runtime flexibility. ASIC approaches are similarly constrained, often being tailored to specific data structures, making them less applicable to a broader range of workloads.

Poor performance/power efficiency in wimpy SmartNICs: Programmable SmartNICs have driven efforts to offload computations to onboard network processors. Some approaches use wimpy processors like ARM or RISC-V for general-purpose computations near memory, but their processing speeds are slower than CPU-based or FPGA-based accelerators. This can make them a performance bottleneck, especially at high memory bandwidth. Moreover, their slower execution results in higher energy per pointer traversal, making them less power-efficient for memory-intensive workloads.

3.3.3 PULSE Overview

Design Overview The framework innovates on three key design elements. Central to the framework's design is its iterator-based programming model that requires minimal effort to port real-world data structure traversals. The framework supports stateful traversals using a scratchpad, where developers can store and update arbitrary intermediate states during the iterator's execution. Properties specific to iterator patterns enable efficient accelerator design and distributed traversals.

The iterator code provided by developers is translated into the framework's instruction set architecture (ISA) to be executed by accelerators. The framework achieves energy efficiency and performance through a novel accelerator that decouples logic and memory pipelines, with an ISA specifically designed for iterator patterns. The accelerator uses a specialized scheduler to ensure high utilization and performance.

The framework also supports scalable distributed pointer traversals by leveraging programmable network switches to reroute requests that must cross memory node boundaries. It employs hierarchical address translation in the network, where memory node-level address translation is performed at the switch, and the memory node accelerator handles local address translation and protection. During traversal, if the memory node accelerator determines the address is not local, it returns the request to the switch, which reroutes it to the correct memory node.

Assumptions: The framework does not offload synchronization to its accelerators but requires the application logic at the CPU node to manage locks for offloaded operations. While recent efforts have enabled locking primitives on NICs and programmable switches, these are orthogonal to our work and can be incorporated into the framework. Lastly, the framework does not innovate on caching but adapts a transparent caching scheme from prior work.

3.3.4 PULSE programming model

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

We analyzed implementations of a wide range of popular data structures to identify common structures in pointer traversals. We found that most traversals: (1) initialize a start pointer using data structure-specific logic, (2) iteratively use data structure-specific logic to determine the next pointer to look up, and (3) check a data structure-specific termination condition at the end of each iteration to determine if the traversal should end. This structure closely resembles the iterator design pattern, which is common across almost all programming languages. Thus, the iterator pattern makes an ideal candidate for the interface between hardware and software layers for pointer traversals.

The interface exposes three functions that must be implemented by the user: (1) init(), which initializes the start pointer, (2) next(), which updates the current pointer to the next pointer in the traversal, and (3) end(), which determines if the traversal should end. The interface then uses these implementations to execute the pointer traversal iteratively using the execute() function. We discuss two key aspects of the iterator abstraction that were necessary to balance expressiveness and limitations for operations on linked data structures.

Stateful Traversals: Pointer traversals in many data structures are stateful, and the state can vary widely. For instance, in hash table lookups, the state is the search key that must be compared against a linked list of keys in a hash bucket. To facilitate this, iterators maintain a scratch_pad that the developer can use to store arbitrary state. The state is initialized in init(), updated in next(), and finalized in end(). Since execute() returns the contents of scratch_pad, developers can place the data that they want to receive in it.

Bounded Computations: The accelerators support only lightweight processing in memory-intensive operations. While init() is executed on the CPU node, next() and end() are offloaded to the accelerators; hence, the framework limits what memory accesses and computations can be performed. Within each iteration, the framework disallows nondeterministic executions, such as unbounded loops. Across iterations, execute() limits the maximum number of iterations that a single request can perform to ensure long traversals do not block other requests. If a request exceeds the maximum iteration count, the framework terminates the traversal and returns the scratch_pad value to the CPU node, which can issue a new request to continue the traversal from that point.

An Illustrative Example: We demonstrate how the find() operation on C++ STL unordered_map can be ported to the framework. The simplified STL implementation computes a hash function, determines a pointer to the hash bucket, and iterates through the linked list, terminating if the key is found or the linked list ends without finding it.

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

3.3.5 Accelerating Pointer Traversals on a Node

3.3.6 Distributed Pointer Traversals

3.3.7 Real-world Applications and Evaluation

3.3.8 Discussion and Conclusion

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 [38] from Nvidia and Compute Express Link (CXL) [4] 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 [39–41], 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 [41,42], 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) [43], 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 [44] 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.

4.1.1 Introduction

4.1.2 Background and Methodology

4.1.3 CXL 1.1 Performance characteristics

4.1.4 Memory Capacity-bound Applications

4.1.5 Memory Bandwidth-bound Applications

4.1.6 Cost Implications

4.1.7 Discussion and Conclusion

Chapter 5

Future Work

Appendix A

Appendix

If you need an appendix, it will go here.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[24] Intel Rack Scale Design: Just what is it? https://www.datacenterdynamics.com/en/opinions/intel-rack-scale-design-just-what-is-it/.

[25] Rack-scale Computing. https://www.microsoft.com/en-us/research/project/rack-scale-computing/.

[26] Terabit Ethernet: The New Hot Trend in Data Centers. https://www.lanner-america.com/blog/terabit-ethernet-new-hot-trend-data-centers/, 2019.

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

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

[29] Disaggregation and Programmable Forwarding Planes. https://www.barefootnetworks.com/blog/disaggregation-and-programmable-forwarding-planes/.

[30] Intel Ethernet Switch FM6000 Series. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[31] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable Packet Scheduling at Line Rate. In *SIGCOMM*, 2016.

[32] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM*, 2013.

[33] MSI Protocol. https://en.wikipedia.org/wiki/MSI_protocol.

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

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

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

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

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

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

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

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

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

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

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