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. Our discussion begins with an outline of the essential requirements for 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.

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.

Performance. As discussed in Section 1, seamlessly transitioning applications to a disaggregated architecture can result in performance issues. One notable concern is that the data structure libraries used by applications may not be aware of the underlying hardware and OS placement policies. Applications with sequential or predictable access patterns may not experience significant performance degradation, provided the OS layer can prefetch data effectively. However, applications with irregular memory access patterns may suffer from frequent back-and-forth data transfers between computation and memory. This problem is exacerbated by slower interconnects (e.g., CXL or network) compared to fast memory buses.

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 Applications and Evaluation

2.6 Related Work

2.7 Conclusion

Chapter 3

Operating System Layer

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

3.2.3 MIND Design

3.2.4 MIND Implementation

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

3.3.2 Motivation

3.3.3 PULSE Overview

3.3.4 PULSE programming model

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× 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.